

Quinta edición

Ingeniería del Software

Un enfoque práctico

ROGER S. PRESSMAN

ADAPTADO POR DARREL INCE



.

CONSULTOR EDITORIAL
ÁREA DE INFORMÁTICA Y COMPUTACIÓN

Gerardo **Quiroz Vieyra**
Ingeniero de Comunicaciones y Electrónica
por la ESIME del Instituto Politécnico Nacional
Profesor de la Universidad Autónoma Metropolitana
Unidad Xochimilco
MÉXICO

www.elsolucionario.net

INGENIERÍA DEL SOFTWARE

UN ENFOQUE PRÁCTICO

Quinta edición

Roger S. Pressman
R.S. Pressman & Associates, Inc.

ADAPTACIÓN
Darrel Ince
Open University

TRADUCCIÓN

Rafael Ojeda Martín **Isabel Morales Jareño**
Virgilio Yagüe Galaup **Salvador Sánchez Alonso**

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software
Facultad de Informática / Escuela Universitaria de Informática
Universidad Pontificia de Salamanca *campus* Madrid (España)

Jorge A. Torres Jiménez
Director de la carrera de Ingeniería de Sistemas Computacionales
Instituto Tecnológico (TEC) de Monterrey *campus* Querétaro (México)

COLABORACIÓN

Óscar San Juan Martínez **Juana González González**
Ricardo Lozano Quesada **Lorena Esmoris Galán**

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software
Facultad de Informática / Escuela Universitaria de Informática
Universidad Pontificia de Salamanca *campus* Madrid (España)

REVISIÓN TÉCNICA
Héctor Castán Rodríguez
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software
Facultad de Informática / Escuela Universitaria de Informática
Universidad Pontificia de Salamanca *campus* Madrid (España)

DIRECCIÓN, COORDINACIÓN Y REVISIÓN TÉCNICA
Luis Joyanes Aguilar
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software
Facultad de Informática / Escuela Universitaria de Informática
Universidad Pontificia de Salamanca *campus* Madrid (España)



MADRID • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MÉXICO
NUEVA YORK • PANAMÁ • SAN JUAN • SANTAFÉ DE BOGOTÁ • SANTIAGO • SÃO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI • PARÍS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO

.

INGENIERÍA DEL SOFTWARE. Un enfoque práctico. (5.ª edición)

McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U.
Edificio Valrealty, 1.ª planta
Basauri, 17
28023 Aravaca (Madrid)

Traducido de la quinta edición en inglés de
SOFTWARE ENGINEERING. A Practitioner's Approach. European Adaptation

ISBN: 0-07-709677-0

ISBN: 84-481-3214-9
Depósito legal: M. 42.852-2001

Editora: Concepción Femández Madrid
Diseño de cubierta: Design Master Dima
Compuesto en FER
Impreso en: Imprenta FARESO. S. A.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

CONTENIDOS A PRIMERA VISTA

CONTENIDO

ACERCA DEL AUTOR

ROGER S. Pressman es una autoridad internacionalmente reconocida en la mejora del proceso del Software y en las tecnologías de la Ingeniería del Software. Durante tres décadas, ha trabajado como ingeniero de Software, gestor, profesor, autor y consultor centrándose en los temas de la Ingeniería del Softwate.

Como especialista **y** gerente industrial, el Dr. Pressman ha trabajado en el desarrollo de sistemas CAD/CAM para aplicaciones avanzadas de ingeniería y fabricación. También ha ocupado puestos de responsabilidad para programación científica y de sistemas.

Después de recibir el título de Doctor en Ciencias Físicas en Ingeniería de la Universidad de Connecticut, el Dr. Pressman se dedicó a la enseñanza donde llegó a ser Profesor Asociado (Bullard Associate Professor) de Informática en la Universidad de Bridgeport **y** Director del Computer-Aided Design an Manufacturing Center en esta Universidad.

El Dr. Pressman es actualmente el Presidente de R. S. Pressman & Associates, Inc., una empresa de asesoría especializada en métodos **y** formación de Ingeniería del Softwate. Trabaja como asesor jefe, y está especializado en la asistencia a compañías para establecer unas prácticas eficientes de la Ingeniería del Software. También ha diseñado y desarrollado productos para la formación en Ingeniería del Softwate de la compañía y de mejora del proceso: *Essential Software Engineering,* un currículum en vídeo totalmente actualizado que se cuenta entre los trabajos industriales más extensos acerca de este tema **y,** *Process Advisor*, un sistema programado para la mejora en el proceso de ingeniería del softwate. Ambos productos son utilizados por cientos de compañías mundiales.

El Dr. Pressman ha escrito numerosos artículos técnicos, participa regularmente en revistas industriales, y ha publicado 6 libros. Además de *Ingeniería del Software: Un Enfoque Práctico,* ha escrito *A Manager's Guide to Software Engineering* (McGraw-Hill), un libro que hace uso de un exclusivo formato de preguntas **y** respuestas para presentar las líneas generales de Administración para implantar y comprender la tecnología; *Making Software Engineering Happen* (Prentice-Hall), que es el primer libro que trata los problemas críticos de administración asociados a la mejora del proceso del Software **y** *Software Shock* (Dorset House), un tratado centrado en el Software y su impacto en los negocios y en la sociedad. El Dr. Pressman es miembro del consejo editorial del *ZEEE Software* **y** del *Cutter IT Journal*, **y** durante muchos años fue editor de la columna «Manager» del *IEEE Software.*

El Dr. Pressman es un conocido orador, impartiendo un gran número de conferencias industriales principalmente. Ha presentado tutoriales en la Conferencia Internacional de Ingeniería del Software **y** en muchas otras conferencias industriales. Es miembro de ACM, IEEE y Tau Beta Pí, Phi Kappa Phi, Eta Kappa Nu y Pi Tau Sigma.

.

PREFACIO

CUANDO un software de computadora se desarrolla con éxito - cuando satisface las necesidades de las personas que lo utilizan; cuando funciona impecablemente durante mucho tiempo; cuando es fácil de modificar o incluso es más fácil de utilizar — puede cambiar todas las cosas y de hecho las cambia para mejor. Ahora bien, cuando un software de computadora falla – cuandolos usuarios no se quedan satisfechos, cuando es propenso a errores; cuando es difícil de cambiar e incluso más difícil de utilizar — pueden ocurrir y de hecho ocurren verdaderos desastres. Todos queremos desarrollar un software que haga bien las cosas, evitando que esas cosas malas merodeen por las sombras de los esfuerzos fracasados. Para tener éxito al diseñar y construir un software necesitaremos disciplina. **Es** decir, necesitaremos un enfoque de ingeniería.

Durante los 20 años que han transcurrido desde que se escribió la primera edición de este libro, la ingeniería del software ha pasado de ser una idea oscura y practicada por un grupo muy pequeño de fanáticos a ser una disciplina legítima de ingeniería. Hoy en día, esta disciplina está reconocida como un tema valioso y digno de ser investigado, de recibir un estudio concienzudo y un debate acalorado. A través de la industria, el título preferido para denominar al «programador» ha sido reemplazado por el de «ingeniero de software». Los modelos de proceso de software, los métodos de ingeniería del software y las herramientas del software se han adaptado con éxito en la enorme gama de aplicaciones de la industria.

Aunque gestores y profesionales reconocen igualmente la necesidad de un enfoque más disciplinado de software, continúan debatiendo sobre la manera en que se va a aplicar esta disciplina. Muchos individuos y compañías todavía desarrollan software de manera algo peligrosa, incluso cuando construyen sistemas para dar servicio a las tecnologías más avanzadas de hoy en día. Muchos profesionales y estudiantes no conocen los métodos nuevos **y,** como resultado, la calidad del software que se produce sufrirá y experimentará malas consecuencias. Además, se puede decir que seguirá existiendo debate y controversia a cerca de la naturaleza del enfoque de la ingeniería del software. El estado de la ingeniería es un estudio con contrastes. Las actitudes han cambiado y se ha progresado, pero todavía queda mucho por hacer antes de que la disciplina alcance una madurez total.

La quinta edición de la *Ingeniería del Software: Un Enfoque Práctico* pretende servir de guía para conseguir una disciplina de ingeniería madura. Esta edición, al igual que las cuatro anteriores, pretende servir a estudiantes y profesionales, y mantiene su encanto como guía para el profesional de industria y como una introducción completa al tema de la ingeniería para alumnos de diplomatura, o para alumnos en primer año de segundo ciclo. El formato y estilo de la quinta edición ha experimentado cambios significativos, con una presentación más agradable y cómoda para el lector, y con un contenido de acceso más fácil.

La quinta edición se puede considerar más que una simple actualización. La revisión de este libro se ha realizado para adaptarse al enorme crecimiento dentro de este campo y para enfatizar las nuevas e importantes prácticas de la ingeniería del software. Además, se ha desarrollado un sitio Web con todo lo necesario y esencial para complementar el contenido del libro. Junto con la quinta edición de *Ingeniería del Software: Un Enfoque Práctico,* la Web proporciona una gama enorme de recursos de ingeniería del software de la que se beneficiarán instructores, estudiantes y profesionales de industria.

La estructura de la quinta edición se ha establecido en cinco partes y la intención ha sido la de realizar una división por temas, y ayudar así a instructores que quizás no tengan tiempo de abarcar todo el libro en un solo trimestre. La Parte Primera, *El producto y el proceso,* es una introducción al entorno de la ingeniería del software. La intención de esta parte es introducir el tema principal y, lo que es más importante, presentar los conceptos necesarios para los capítulos siguientes. La Parte Segunda, *Gestión de proyectos de software*, presenta los temas relevantes para planificar, gestionar y controlar un proyecto de desarrollo de ingeniería. La Parte Tercera, *Métodos convencionales para la ingeniería del software,* presenta los métodos clásicos de análisis, diseño y pruebas considerados como la escuela «convencional» de la ingeniería del software. La Parte Cuarta, *Ingeniería del software orientada a objetos,* presenta los

métodos orientados a objetos a lo largo de todo el proceso de ingeniería del software, entre los que se incluyen análisis, diseño y pruebas. La Parte Quinta, *Temas avanzados en ingeniería del software*, introduce los capítulos especializados en métodos formales, en ingeniería del software de sala limpia, ingeniería del software basada en componentes, ingeniería del software cliente/servidor, ingeniería de Web, reingeniería y herramientas CASE.

La estructura de la quinta edición en cinco partes permite que el instructor «agrupe» los temas en función del tiempo disponible y de las necesidades del estudiante. Un trimestre completo se puede organizar en tomo a una de las cinco partes. Por ejemplo, un «curso de diseño» podría centrarse solo en la Parte Tercera o Cuarta; un «curso de métodos» podría presentar los capítulos seleccionados de las Partes Tercera, Cuarta y Quinta. Un «curso de gestión» haría hincapié en las Partes Primera y Segunda. Con la organización de esta quinta edición, he intentado proporcionar diferentes opciones para que el instructor pueda utilizarlo en sus clases.

El trabajo que se ha desarrollado en las cinco ediciones de *Ingeniería del Software: Un Enfoque Práctico* es el proyecto técnico de toda una vida. Los momentos de interrupción los he dedicado a recoger y organizar información de diferentes fuentes técnicas. Por esta razón, dedicaré mis agradecimientos a muchos autores de libros, trabajos y artículos, así como a la nueva generación de colaboradores en medios electrónicos (grupos de noticias, boletines informativos electrónicos y World Wide Web), quienes durante 20 años me han ido facilitando otras visiones, ideas, y comentarios. En cada uno de los capítulos aparecen referencias a todos ellos. Todos están acreditados en cuanto a **su** contribución en la rápida evolución de este campo. También quiero dedicar mis agradecimientos a los revisores de esta quinta edición. **Sus** comentarios y críticas son de un valor incalculable. Y por Último dedicar un agradecimiento y reconocimiento especiales a Bruce Maxim de la Universidad de Michigan —DearBorn, quien me ayudó a desarrollar el sitio Web que acompaña este libro, y como persona responsable de una gran parte del diseño y contenido pedadógico—.

El contenido de la quinta edición de *Ingeniería del Software: Un Enfoque Práctico* ha tomado forma gracias a profesionales de industria, profesores de universidad y estudiantes que ya habían utilizado las ediciones anteriores, **y** que han dedicado mucho tiempo en mandarme sus sugerencias, críticas e ideas. Muchas gracias también a todos vosotros. Además de mis agradecimientos personales a muchos clientes de industria de todo el mundo, quienes ciertamente me enseñan mucho **más** de lo que yo mismo puedo enseñarles.

A medida que he ido realizando todas las ediciones del libro, también han ido creciendo y madurando mis hijos Mathew y Michael. **Su** propia madurez, carácter y éxito en la vida han sido mi inspiración. Nada me ha llenado con más orgullo. Y, finalmente, a Bárbara, mi cariño y agradecimiento por animarme a elaborar esta quinta edición.

Roger S. Pressman

EL libro de Roger Pressman sobre Ingeniería del software es verdaderamente excelente: Siempre he admirado la profundidad del contenido y la habilidad del autor para describir datos difíciles de forma muy clara. De manera que tuve el honor de que McGraw-Hill me pidiera elaborar la Adaptación Europea de esta quinta edición. Esta es la tercera adaptación, y su contenido es un acopio de la quinta edición americana y el material que yo mismo he escrito para ofrecer una dimensión europea.

Las áreas del libro que contienen ese material extra son las siguientes:

- Existe una gran cantidad de material sobre métodos formales de desarrollo del software. Estas técnicas fueron pioneras en el Reino Unido y Alemania, y han llegado a formar parte de los juegos de herramientas críticos para los ingenieros de software en el desarrollo de sistemas altamente integrados y críticos para la seguridad.
- He incluido una sección sobre patrones de diseño. Estos son los que tienen lugar generalmente en miniarquitecturas que se dan una y otra vez en sistemas orientados a objetos, y que representan plantillas de diseño que se reutilizarán una y otra vez. He viajado por toda Europa, y me he encontrado constantemente compañías cuyo trabajo depende realmente de esta tecnología.
- He aportado material sobre métricas y en particular la utilización de GQM como métrica de método de desarrollo. A medida que la ingeniería del software se va integrando poco a poco dentro de una disciplina de ingeniería, esta tecnología se va convirtiendo en uno de sus fundamentos. La métrica ha sido uno de los puntos fuertes en Europa de manera que espero que toda la dedicación que he puesto en este tema lo refleje.
- He incluido también material sobre el Lenguaje de Modelado Unificado (UML) el cual refleja el gran incremento de utilización de este conjunto de notaciones en Europa Occidental. Además, sospecho que van a ser de hecho las notaciones de desarrollo de software estándar durante los próximos tres o cuatro años.
- He dado mayor énfasis al desarrollo de sistemas distribuidos mediante el lenguaje de programación Java para ilustrar la implicación de algunos de los códigos. Ahora que Internet y el comercio electrónico están en pleno auge en Europa, las compañías cada vez se vuelcan más en las técnicas de ingeniería del software al desarrollar aplicaciones distribuidas. Y esto también queda reflejado en el libro.
- He incluido también material sobre métodos de seguridad e ingeniería. Con la utilización de Intemet (una red abierta) todos los ingenieros del software tendrán que saber muchas más técnicas tales como firmas digitales y criptografía.
- Hacia el final del libro he hecho especial hincapié en la utilización de aplicaciones de comercio electrónico para mostrar de esta manera la tecnología distribuida.

Existen dos partes que hacen referencia al libro: un sitio Web americano muy importante, desarrollado por el Dr. Pressman; y un sitio de entrada, cuya dirección se proporciona a lo largo de todo el libro al final de cada capítulo. Este sitio contiene el material relevante para la edición europea y los enlaces con el sitio americano. Ambos sitios Web en combinación contienen sub-Webs con recur*sos* para Estudiantes, para Profesores y para Profesionales.

En los *Recursos para el estudiante* se incluye una guía de estudio que resume los puntos clave del libro, una serie de autopruebas que permiten comprobar la comprensión del material, cientos de enlaces a los recursos de ingeniería del software, un caso práctico, materiales complementarios y un tablón de mensajes que permite comunicarse con otros lectores.

En la parte *Recursos para el profesor* se incluye una guía para el instructor, un conjunto de presentaciones en PowerPoint basadas en este libro, un conjunto de preguntas que se pueden utilizar para practicar mediante deberes y exámenes, un conjunto de herramientas muy pequeñas (herramientas sencillas de ingeniería del software adecuadas para su utilización en clase), una herramienta de Web que permite crear un sitio Web específico para un curso, y un tablón de mensajes que hace posible la comunicación con otros instructores.

En los *Recursos para profesionales* se incluye documentación para los procesos de ingeniería del software, listas de comprobación para actividades tales como revisiones, enlaces a proveedores de herramientas profesionales, cientos de enlaces a recursos de ingeniería del software, información sobre los paquetes de vídeo de Pressman, y comentarios y ensayos industriales acerca de varios temas de la ingeniería del software.

Ingeniería del software es un libro excelente, y espero que los aportes que he incluido para el lector europeo hagan de este libro una lectura obligada.

Darrel Ince

.

PRÓLOGO A LA CUARTA EDICIÓN EN ESPAÑOL

EL ESTADO DEL ARTE DE LA INGENIERÍA DEL SOFTWARE

La **Ingeniería de/l Software**[1] es una disciplina o área de la Informática o Ciencias de la Computación, que ofrece métodos y técnicas para desarrollar y mantener software de calidad que resuelven problemas de todo tipo. Hoy día es cada vez más frecuente la consideración de la ***Zngenierh de/l Software*** como una nueva área de la ingeniería, y el ***ingeniero de/l software*** comienza a ser una profesión implantada en el mundo laboral internacional, con derechos, deberes y responsabilidades que cumplir, junto a una, *ya,* reconocida consideración social en el mundo empresarial y, por suerte, para esas personas con brillante futuro.

La Ingeniería de/l Software trata con áreas muy diversas de la informática y de las ciencias de la computación, tales como construcción de compiladores, sistemas operativos o desarrollos en Intranet/Internet, abordando todas las fases del ciclo de vida del desarrollo de cualquier tipo de sistemas de información y aplicables a una infinidad de áreas tales como: negocios, investigación científica, medicina, producción, logística, banca, control de tráfico, meteorología, el mundo del derecho, la red de redes Intemet, redes Intranet y Extranet, etc.

Definición del término «Ingeniería de/l Software»

El término ***Zngenierh*** se define en el **DRAE**[2] como: «**1**. Conjunto de conocimientos y técnicas que permiten aplicar el saber científico a la utilización de la materia y de las fuentes de energía. **2.** Profesión y ejercicio del ingeniero» y el término ***ingeniero*** se define como «**1.** Persona que profesa o ejerce la ingeniería». De igual modo la Real Academia de Ciencias Exactas, Físicas y Naturales de España define el término ***Ingeniería*** como: «Conjunto de conocimientos y técnicas cuya aplicación permite la utilización racional de los materiales y de los recursos naturales, mediante invenciones, construcciones u otras realizaciones provechosas para el hombre»[3].

Evidentemente, si la Ingeniería del Software es una nueva ingeniería, parece lógico que reúna las propiedades citadas en las definiciones anteriores. Sin embargo, ni el DRAE ni la Real Academia Española de Ciencias han incluido todavía el término en sus Últimas ediciones; en consecuencia vamos a recurrir para su definición más precisa a algunos de los autores más acreditados que comenzaron en su momento a utilizar el término o bien en las definiciones dadas por organismos internacionales profesionales de prestigio tales como IEEE o ACM. Así, hemos seleccionado las siguientes definiciones de ***Zngenierh del Software:***

Definición 1

Ingeniería de Software es el estudio de los *principios* y *metodologías* para desarrollo y mantenimiento de sistemas de software [Zelkovitz, 1978][4].

Definición 2

Ingeniería del Software es la aplicación *práctica* del conocimiento científico en el diseño y construcción de programas de computadora y la *documentación* asociada requerida para desarrollar, operar (funcionar) y mantenerlos. Se conoce también como desarrollo de software o producción de software [Bohem, 19761[5].

[1] En Hispanoamérica. el término utilizado normalmente es: Ingeniería de Software.

[2] DRAE, Diccionario de la Real Academia Española de la Lengua.

[3] Vocabulario Científico y Técnico, edición de 1996.

[4] ZELKOVITZ, M. V., CHAW, A. C. y GANNON, J. D.: *Principles of Software Engineering* and *Design.* Prentice-Hall, Englewoods Ciif, 1979.

[5] BOEHM, B. W.: «Software Engineering», *IEEE Transactions* on *Computers,* C-25, núm. 12, diciembre, pp. 1226-1241.

Definición 3

Ingeniería del Software trata del establecimiento de los principios y métodos de la ingeniería a fin de obtener software de modo rentable que sea *fiable* y trabaje en *máquinas reales* [Bauer, 1972][6].

Definición 4

La aplicación de *un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación* (funcionamiento) y mantenimiento del software; es decir, la aplicación de ingeniería al software. 2. El estudio de enfoques como en (1) [IEEE, 1993][7].

Tomando como base la referencia de estas definiciones seleccionadas, se producen de inmediato las preguntas: *¿cuáles son las actividades que se encuadran hoy día en el mundo de la ingeniería del software?* y *¿cómo se enseña la disciplina de ingeniería del software en las universidades, escuelas de ingenieros e institutos tecnológicos?*

La respuesta a la primera pregunta se manifiesta de muy diversas formas, pero creemos que tal vez las fuentes más objetivas sean las conferencias, eventos y acontecimientos internacionales más relevantes realizados en estos Últimos años. Así, de los estudiados, hemos considerado como congresos significativos, los convocados por **SIGSOFT** (Special Interest Group on Software Engineering) de la **ACM**[8]: Intemational Conference on Software Engineering **(ICSE,** copatrocinada con IEEE) celebrada en Boston, Massachussets, USA (17-23 de mayo de 1997) y la próxima conferencia anunciada para celebrarse en 1998 en Kyoto, Japón **(ICSE,** 19-25 de abril de 1998); *5*." Symposium Foundations of Software Engineering, **SIGSOFT 97** (Zurich, 22-25 septiembre de 1997) y 6." Symposium **SIGSOFT 98** (Orlando, Florida, USA, 3-5 noviembre de 1998).

En los congresos citados anteriormente y en algunas otras fuentes como revistas de ACM/IEE y otras de tipo profesional o comercial específicas de ingeniería de software, hemos analizado sus programas, tutoriales, talleres de trabajo, contenidos, etc., y hemos seleccionado una lista con los temas más candentes del actual *estado del arte de la ingeniería del Software.* Los temas más sobresalientes son:

- Inspección de software crítico.
- Software de Tecnologías de Procesos de Negocios.
- Arquitecturas de Software Distribuido.
- Introducción a UML (Metodología de objetos, método unificado de Booch, Rumbaugh y Jacobson).
- Control técnico de proyectos software.
- Marcos de trabajo (*frameworks*) de empresa orientados a objetos.
- Una introducción a CORBA (Estándar para objetos distribuidos).
- Estrategias de ingeniería inversa para migración de software.
- Ingeniería de objetos.
- Modelado y análisis de arquitectura de software.
- Objetos distribuidos.
- Sistemas Cliente/Servidor.
- Reingeniería.
- CASE.
- Análisis y Diseño Orientados a Objetos.
- **...**

Esta cuarta edición ha mejorado en cantidad y calidad a la tercera edición, pero su actualidad en el año 1997, reside en el hecho de que la mayoría de temas tratados o que se van a tratar en los congresos de la ACM (listados anteriormente), están contemplados y tratados por el autor en este libro. En cualquier forma, deseamos destacar muy expresamente algunas mejoras concretas que vienen a llenar una importante laguna que existía en los libros de ingeniería del

[6] BAUER, F. L.: «Software Engineering», *Information Processing,* 71, North Holland Pubiishing Co., Amsterdarn, 1972
[7] IEEE: *Standards Collection: Software Engineering,* IEEE Standard 610.12-1990, IEEE, 1993.
[8] URL de **ACM,** http://www.acm.org.

software en inglés y, por supuesto, en español. Así, destacamos el estudio y profundidad que se hace de temas tan candentes y de actualidad en el mundo de la ingeniería del software tales como: *Métodosformales, reutilización de software, reingeniería, métodos de software Cliente/Servidor,* ***CASE,*** *análisisldiseñolpruebas* y *métricas orientados a objetos, etc.,* junto con un epílogo donde muestra el estado actual y futuro de la Ingeniería del Software. Con relación a la tercera edición se aprecia una consolidación de tratamientos y una unificación de bloques de conocimiento que consiguen mejorar el aprendizaje del lector.

Una de las aportaciones más relevantes que aporta esta nueva edición es la excelente bibliografía y referencias bibliográficas que se adjuntan en cada capítulo del libro, junto a una novedad que poco a poco comienza a incorporarse en las buenas obras científicas y que aquí alcanza un excelente nivel: ***direcciones electrónicas (URLs) de sitios Webde Internet*** notables, donde se pueden ampliar conocimientos, fuentes bibliográficas, consultorías, empresas, organismos internacionales, etc., especializados en Ingeniería de Software.

En lo relativo a la segunda pregunta antes enunciada, su respuesta implica el uso de libros de texto y manuales que ayuden al estudiante a complementar las lecciones impartidas por el profesor, así como preparar sus trabajos, prácticas, exámenes, etc. Un libro para ser considerado de texto o referencia de una determinada asignatura requiere que su contenido contenga todos o la mayoría de los descriptores o tópicos considerados en el programa de la asignatura. Veamos por qué esta obra es idónea como libro de texto para asignaturas del cum'culum universitario de ***Ingeniería de/l Software.***

EL LIBRO COMO TEXTO DE REFERENCIA UNIVERSITARIA

La importancia fundamental de la disciplina *Ingeniería del Software* se está manifestando, de modo destacado, en los *currículum* de informática y ciencias de la computación de la mayoría de las universidades de todo el mundo, y seguirá creciendo su importancia a medida que nos acerquemos al tercer milenio.

Debido a estas circunstancias, las organizaciones profesionales, los departamentos educativos de los diversos gobiernos y los departamentos universitarios se han preocupado en esta década y en las anteriores de estandarizar los programas curriculares de las diferentes carreras, incluyendo materias (asignaturas) troncales y obligatorias, en los planes de estudios de Facultades y Escuelas de Ingenieros de todo el mundo.

El caso más significativo lo constituyen las organizaciones profesionales internacionales que se han preocupado también de este proceso. Entre las más destacadas sobresalen ACM (Association of Computing Machinery) e IEEE (Institute for Electrical and Electronics Engineers). Así, en el año 1991, estas dos organizaciones publicaron conjuntamente unas recomendaciones con los requisitos (materias) imprescindibles que, al menos, debían contemplar todos los planes de estudios de carreras relacionadas con Ciencias de la Computación (Informática). Estas recomendaciones han sido seguidas por todas las universidades de Estados Unidos y prácticamente, de una u otra forma, por casi todas las universidades europeas e hispanoamericanas, desde el año de su publicación.

Las recomendaciones ACM/IEEE[9] dividen los requisitos del currículum en nueve áreas diferentes, con subdivisiones en esas áreas. Para cada subárea se recomienda un número mínimo de unidades de conocimiento *(knowledge units*) con una indicación de tiempo para cada una de ellas (períodos de 50 minutos/clase). En estas nueve áreas se incluyen: Algoritmos, Arquitectura, Inteligencia Artificial, Bases de Datos, Interfaces Hombre/Máquina, Computación numérica, Sistemas Operativos, Programación, Ingeniería del Software, Lenguajes de Programación y Temas legales, profesionales y sociales. Los temas recomendados en el área de Ingeniería de Software son:

1. Conceptos fundamentales de resolución de problemas.
2. Proceso de desarrollo de software.
3. Especificaciones y requisitos de software.
4. Diseño e implementación de software.
5. Verificación y validación.

[9] *http:/www acm.org ; http:/xavier.xu.edu:8000/~~lewan/new cum'culum.html*

En España, el Consejo de Universidades, organismo encargado de dictar directrices y normas para los planes de estudios de las universidades tiene redactadas las normativas que han de cumplir todas las universidades que deseen impartir las carreras de Ingeniería en Informática (cinco años académicos, diez semestres), Ingeniería Técnica en Informática de Sistemas e Ingeniería Técnica en Informática de Gestión (tres años académicos, seis semestres) y se publican oficialmente en el *Boletín Oficial del Estado (BOE)*. En estas normativas de obligado cumplimiento por todas las universidades, se incluyen las materias (asignaturas o conjunto de asignaturas) troncales que se deben incluir obligatoriamente en todos los planes de estudios, además de otras asignaturas con carácter obligatorio y opcional que cada universidad fija en sus planes de estudios.

Ingeniería del Software es una materia troncal incluida en las carreras citadas. 18 créditos (cada crédito equivale a diez horas de clase) en la ingeniería superior *(ingeniero informútico)* y 12 créditos en la ingeniería técnica de informática de gestión *(ingeniero técnico informútico)*. Esto significa una carga lectiva considerable que todos los estudiantes de carreras de informática y de computación han de estudiar, normalmente en los cursos 3.° a 5.°

Este libro puede ser utilizado en diferentes tipos de cursos de ingeniería de software tanto a nivel universitario como a nivel profesional:

1. Cursos introductorios de Ingeniería del Software. Estudiantes de primer ciclo (tres años) y segundo ciclo (cinco años), o en carreras de cuatro años (como suele ser el caso de las universidades hispanoamericanas y alguna española), sin experiencia previa en Ingeniería del Software, pero con formación de dos o tres cursos universitarios (cuatro o cinco semestres) al menos.
2. Cursos introductorios y de nivel medio sobre temas específicos de Ingeniería del Software tales como análisis de requisitos y especificaciones, gestión de proyectos de software, métodos convencionales de Ingeniería del Software, etc.
3. Cursos avanzados en temas de Ingeniería del Software tales como: análisis y diseño orientados a objetos, métricas, ingeniería inversa, Cliente/Servidor, Reutilización de software, etcétera.

Este libro explica todos los temas incluidos en la asignatura o materia troncal ***Ingeniería del Software*** por el Consejo de Universidades de España, así como las unidades SE2 a SE5 (la unidad SE1 se refiere a conceptos de tipo general que suelen incluirse en otras asignaturas básicas) del currículum de la ACM/IEEE de 1991. Por nuestro conocimiento personal (conferencias, cursillos, estancias...) de muchas universidades hispanoamericanas, nos consta que los planes de estudio de estas universidades incluyen también asignaturas de tipo obligatorio de Ingeniería del Software que siguen prácticamente las recomendaciones de ACM/IEEE y son muy similares a los programas que se siguen también en España.

APÉNDICE

La obra actual incluye gran cantidad de siglas y acrónimos en inglés, la mayoría de las cuales, exceptuando las ya acreditadas en inglés cómo BPR..., se han traducido al español. Por su especial importancia y la gran cantidad de ellas incluidas en el libro, el equipo de traducción decidimos recopilar todas las siglas en inglés y sus traducciones al español; a continuación se ha construido dos tablas ordenadas alfabéticamente en inglés y en español, con el objetivo principal de que el lector pueda localizar fácilmente cualquiera de las siglas que aparecen en el texto, y en su caso, la traducción al español. Estas tablas se presentaron a la editora de la obra que tras ver el servicio que proporcionaba al lector, aceptó incluirlas como Apéndice. De este modo, el lector podrá comprobar en cualquier momento entradas de siglas tanto en español como en inglés.

EPÍLOGO

La traducción de esta obra ha sido un esfuerzo común que hemos realizado profesores de diferentes universidades españolas e hispanoamericanas —profesores de Ingeniería del Software que hemos utilizado, en la mayoría de los casos, las ediciones anteriores de esta obra como libro de referencia en nuestras clases y continuaremos utilizándolo. Por esta circunstancia, hemos

podido apreciar que esta cuarta edición ha mejorado de modo muy notable a las ediciones anteriores y tenemos el convencimiento de que los principios y conceptos considerados en ella, seguirán teniendo una influencia considerable en numerosos estudiantes de ingeniería, licenciatura informática o sistemas computacionales de habla española, como nos consta que siguen influyendo en el mundo de habla inglesa. Estamos seguros de que serán muchos los estudiantes que seguirán formándose en Ingeniería del Software con este libro como referencia fundamental.

Esta cuarta edición llena numerosas lagunas en la literatura de habla española, ya que actualiza los contenidos tradicionales de la Ingeniería del Software y los extiende a los temas avanzados modernos que ya hemos considerado. El excelente trabajo de Roger S. Pressman permitirá seguir formando numerosos y buenos profesionales de Ingeniería del Software para que esta industria pueda seguir desarrollándose.

Madrid, septiembre de 1997

Luis Joyanes Aguilar
Director del Departamento de Lenguajes y Sistemas Informáticos
e Ingeniería del Software
Facultad de Informática y Escuela Universitaria de Informática
Universidad Pontificia de Salamanca. Campus de Madrid

PRÓLOGO A LA QUINTA EDICIÓN EN ESPAÑOL

EL siglo XXI se enfrenta a la creciente implantación de la sociedad del conocimiento. La era del conocimiento en que vivimos no sólo está cambiando la sociedad en sí misma, sino que los nuevos modelos de negocios requieren la reformulación de nuevos conceptos. Conocimiento, activos intangibles, Web, etc., son algunos de los términos más utilizados en cualquier ambiente o negociación. Esta era del conocimiento requiere de nuevas tendencias apoyadas precisamente en el conocimiento. La ingeniería del software no es una excepción, y por ello se requiere no sólo una actualización de conceptos, sino también una comprensión y una formulación del nuevo conocimiento existente en torno a las nuevas innovaciones y teorías de dicha disciplina.

En cada edición de su clásica obra, Roger Pressman nos tiene acostumbrados a la sorpresa y a la admiración por la clara y excelente exposición de los temas tratados. Esta vez tampoco ha sido una excepción, muy al contrario, Pressman da la sensación de haber conseguido «la cuadratura del círculo» o mejor aún, ha encontrado la piedra filosofal para formar y educar a los actuales y —sobre todo— futuros ingenieros de software del futuro (o a los ingenieros informáticos e ingenieros de sistemas y licenciados en informática que se forman en esta disciplina). En esta quinta edición, Pressman proporciona al lector una ingente cantidad de conocimiento relativo a ingeniería del software que facilitará considerablemente su formación con todo el rigor profesional y científico que requiere la nueva era del conocimiento que viviremos en esta década.

EL NUEVO CONTENIDO

Una de las grandes y atractivas novedades de esta quinta edición es su nuevo formato y estilo. El SEPA 5/2, como se le conoce en la versión en inglés, ha mejorado el libro y lo ha hecho más legible y atractivo al lector. Mediante iconos y una lista normalizada de seis cuestiones clave, Pressman va guiando al lector sobre los temas mas importantes de cada capítulo a la vez que su lectura le introduce paulatina e inteligentemente en las ideas y conceptos más importantes. Esta quinta edición contiene todos los temas importantes de la cuarta edición e introduce una gran cantidad de temas relativos a las nuevas tendencias, herramientas y metodologías que plantea la ingeniería de software actual y futura, así como la naciente nueva ingeniería Web. Un estudio detenido del contenido nos conduce a los cambios más sobresalientes realizados en esta quinta edición, que son, entre otros, los siguientes:

- Cinco nuevos capítulos (Capítulo 14, Diseño arquitectónico; Capítulo 15, Diseño de la interfaz de usuario, proporcionando reglas de diseño, procesos de modelado de interfaces, diseño de tareas y métodos de evaluación; Capítulo 16, Diseño a nivel de componentes; Capítulo 27, examina los procesos y la tecnología de la ingeniería de software basada en componentes, y, Capítulo 29, que presenta los nuevos conceptos de Ingeniería Web (procesos WebE, análisis y formulación de aplicaciones Web, es decir arquitectura, navegación y diseño de interfaces, pruebas y aplicaciones Web y gestión de proyectos de ingeniería Web).
- Gran cantidad de actualizaciones y revisiones de los 32 capítulos de su contenido total. Los cambios clave son numerosos, y los más sobresalientes son:

—Modelos de procesos evolutivos (WinWin) y de ingeniería basada en componentes.
—Nuevas secciones sobre control estadístico de la calidad.
—Modelo de estimación de COCOMO 11.
—Técnicas a prueba de errores para gestión de calidad de software **(SQA).**
—Ingeniería de requisitos.
—El lenguaje unificado de modelado, UML (Unified Modeling Language).
—Nuevas regias y teoría de calidad de software que siguen la normativa ISO 9000.

NUEVOS RECURSOS DOCENTES Y PROFESIONALES

Si la edición en papel que tiene el lector en sus manos ya es de por sí excelente, el sitio Web del libro no le queda a la zaga (*www.pressman5.com*). Este sitio es una de las mejores herramientas de las que pueden disponer estudiantes, profesores y maestros, y profesionales del mundo del software. Destaquemos algunas.

Recursos de estudiantes

- Guía de estudios.
- Autotest.
- Recursos basados en la Web.
- Estudio de casos.
- Vídeos.
- Contenidos suplementarios.
- Foros para intercambios de mensajes entre lectores.

Recursos de profesores y maestros

- Guía del profesor.
- Transparencias (acetatos) en PowerPoint.
- Bancos de prueba.
- Herramientas de ingeniería de software.
- Foros de intercambio de mensajes entre colegas.

Recursos profesionales

- Plantillas de productos de documentos y de trabajos.
- Listas de pruebas de ingeniería de software.
- Herramientas de ingeniería de software.
- Herramientas CASE.
- Recursos de ingeniería de software.
- Modelos de procesos adaptables.
- Currículum de vídeos de calidad para la industria.
- Comentarios de la industria.
- Foros profesionales.

EL GLOSARIO DE SIGLAS: UN NUEVO APÉNDICE

Una de las características más sobresalientes de esta obra es que recoge con gran profusión la ingente cantidad de siglas que hoy día se utilizan en la industria del software junto a otras muchas más acuñadas por el propio autor.

El equipo de profesores que ha traducido y adaptado la versión en inglés de común acuerdo con la editora acordó realizar un apéndice que incluyera todas las siglas incluidas en el libro y las traducciones correspondientes en español, y viceversa. Este apéndice busca, al igual que ya se hiciera en la segunda edición en español, facilitar al lector la lectura y seguimiento del modo más fácil posible y que le permita hacer la correspondencia entre ambos idiomas cuado lo necesite. Por ello, este apéndice contiene un diccionario inglés-español y otro español-inglés de siglas. El método que se ha seguido durante la traducción ha sido traducir prácticamente todas las siglas, y sólo se han realizado algunas excepciones, como **SQA** (Software Quality Assurement) por su uso frecuente en la jerga informática, aunque en este caso hemos utilizado ambos términos (en inglés, **SQA** y en español, **GCS,** Gestión de Calidad del Software). En este caso, estas siglas en español coinciden con Gestión de Configuración del Software **(GCS),** por lo que a veces estas siglas se han traducido por GCVS (Gestión de Configuración de Versiones de Soft-

ware) para evitar duplicidades. Ésta es una de las razones fundamentales por la que hemos incluido el glosario de siglas.

EL EQUIPO TRADUCTOR

La edición británica ha sido adaptada por un prestigioso profesor británico, y la edición y la adaptación españolas han sido realizadas por un numeroso equipo de profesores del *Departamento de Lenguajes y Sistemas Informáticos e ingeniería del Software* de la *Facultad de informática y Escuela Universitaria de Informática* de la **Universidad Pontificia de Salamanca (España)** del *campus* de Madrid, que ha contado con la inestimable colaboración de profesores del prestigioso **Instituto Tecnológico (TEC) de Monterrey,** de su *campus* de Querétaro (México). Una obra de la envergadura de esta quinta edición requería —como ya sucedió tambien en la edición anterior— del trabajo y coordinación de numerosas personas. Muchas han sido las horas dedicadas a la traducción, adaptación y sucesivas revisones de galeradas de las pruebas de imprenta. Confiamos haber cumplido nuestra obligación con dignidad y profesionalidad.

EL FUTURO ANUNCIADO

Esta quinta edición ha sido actualizada sustancialmente con nuevos materiales que reflejan el estado del arte de la ingeniería del software. El material obsoleto se ha eliminado de esta edición para crear un texto fácil de leer y entender, **y** totalmente actualizado. Sin embargo, y con ser importante todo su vasto y excelente contenido, queremos destacar que esta nueva edición nos brinda y abre el camino al futuro que señala la moderna ingeniería de software basada en objetos y componentes, así como a la ingeniería Web, conservando el rigor y la madurez de las teorías ya clásicas de la ingeniería del software.

Sin lugar a dudas, Pressman ha conseguido una excelente obra, y en una prueba clara de profesionalidad y excelencia ha logrado superar sus cuatro ediciones anteriores ya de por **sí** excelentes.

Madrid y Carchelejo (España), verano de 2001

Luis Joyanes Aguilar
Director del Departamento de Lenguajes, Sistemas informáticos e Ingeniería de Software
Facultad de Informática/Escuela Universitaria de Informática
Universidad Pontificia de Salamanca *campus* Madrid

UTILIZACIÓN DEL LIBRO

LA quinta edición de *Ingeniería del Software: Un Enfoque Práctico* se ha vuelto a diseñar para aumentar la experiencia del lector y para proporcionar enlaces integrados al sitio Web, **http://www.pressman5.com.** Dentro de este sitio los lectores encontrarán información complementaria útil y rica, y una gran cantidad de recursos para los instructores que hayan optado por este libro de texto en clase.

A lo largo de todo el libro se van encontrando iconos que deberán interpretarse de la siguiente manera:



Utilizado para enfarizar un punto importante en el cuerpo del texto.

El icono de **punto clave** ayudará a encontrar de forma rápida los puntos importantes.



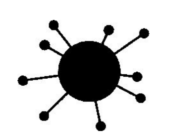
Para punteros que conducirán directamente a los recursos de la Web.

El icono **Referencia web** proporciona punteros directos a sitios Web importantes relacionados con la ingeniería del software.



Un consejo práctico del mundo real aplicado a la ingeniería del software.

El icono de **consejo** proporciona una guía pragmática que servirá de ayuda para tomar la decisión adecuada o evitar los problemas típicos al elaborar software.

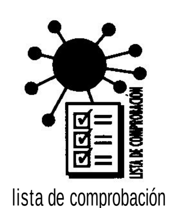


Un tema seleccionado.

El icono de **sitio Web** indica que se dispone de más información sobre el tema destacado en el sitio Web SEPA.



El icono de **signo de interrogación** formula preguntas que se responderán en el cuerpo del texto.



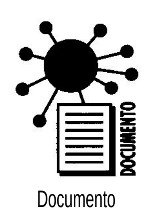
lista de comprobación

El icono de **lista de comprobación** nos señala las listas de comprobación que ayudan a evaluar el trabajo de ingeniería del software que se está llevando a cabo y los productos del trabajo.

Referencia cruzada

Proporciona una referencia cruzada importante dentro del libro

El icono de **referencia cruzada** nos conducirá a alguna otra parte del texto en donde se podrá encontrar información relevante sobre el tema en cuestión.



Documento

El icono de **documento** nos señala los bocetos, descripciones y ejemplos de documentos del sitio Web SEPA.



El icono de **cita** presenta citas interesantes que tienen gran relevancia para los temas que se estén tratando.

.

PARTE

I

EL PRODUCTO Y EL PROCESO

EN esta parte de *Ingeniería del software: un enfoque práctico* aprenderá sobre el producto que va a ser tratado con ingeniería y el proceso que proporciona un marco de trabajo para la tecnología de Ingeniería del software. En los capítulos siguientes se tratan las preguntas:

- ¿Qué es realmente el software de computadora?
- ¿Por qué se lucha para construir sistemas de alta calidad basados en computadoras?
- ¿Cómo se pueden establecer categorías de dominios de aplicaciones para software de computadoras?
- ¿Qué mitos de software van a existir?
- ¿Qué es un «proceso» de software?
- ¿Existe una forma genérica de evaluar la calidad de un proceso?
- ¿Qué modelos de procesos se pueden aplicar al desarrollo del software?
- ¿En que difieren los modelos de proceso lineales e iterativos?
- ¿Cuáles son sus puntos fuertes y débiles?
- ¿Qué modelos de proceso avanzados se han propuesto para la ingeniería del software?

Una vez contestadas todas estas preguntas, estará más preparado para comprender los aspectos técnicos y **de** gestión de la disciplina de ingeniería a la que se dedica el resto del libro.

CAPÍTULO

1 EL PRODUCTO

LAS alarmas comenzaron más de una década antes del acontecimiento. Con menos de dos años a la fecha señalada, los medios de comunicación recogieron la historia. Los oficiales del gobierno expresaron su preocupación, los directores de la industria y de los negocios comprometieron grandes cantidades de dinero, y por Último, las advertencias horribles de catástrofe llegaron a la conciencia del público. El software, al igual que el ahora famoso error Y2K, podría fallar, y como resultado, detener el mundo como nosotros lo conocimos.

Como vimos durante los últimos meses del año 1999, sin querer, no puedo dejar de pensar en el párrafo profético contenido en la primera página de la cuarta edición de este libro. Decía:

> El software de computadora se ha convertido en el ***alma*** *mater*. Es la máquina que conduce a la toma de decisiones comerciales. Sirve de base para la investigación científica moderna y de resolución de problemas de ingeniería. Es el factor clave que diferencia los productos y servicios modernos. Está inmerso en sistemas de todo tipo: de transportes, médicos, de telecomunicaciones, militares, procesos industriales, entretenimientos, productos de oficina..., la lista es casi interminable. El software **es** casi ineludible en un mundo moderno. **A** medida que nos adentremos en el siglo XXI, será el que nos conduzca a nuevos avances en todo, desde la educación elemental a la ingeniería genética.

VISTAZO RÁPIDO

¿Qué es? El software de computadora es el producto que diseñan y construyen los ingenieros del software. Esto abarca programas que se ejecutan dentro de una computadora de cualquier tamaño y arquitectura, documentos que comprenden formularios virtuales e impresos y datos que combinan números y texto y también incluyen representaciones de información de audio, vídeo e imágenes.

¿Quién lo hace? Los ingenieros de software lo construyen, y virtualmente cualquier persona en el mundo industrializado lo utiliza bien directa o indirectamente.

¿Por qué es importante? Porque afecta muy de cerca a cualquier aspecto de nuestra vida y está muy extendido en nuestro comercio, cultura y en nuestras actividades cotidianas.

¿Cuáles son los pasos? Construir software de computadora como construimos cualquier otro producto satisfactorio, aplicando un proceso que conduce a un resultado de alta calidad que satisface las necesidades de la gente que usará el producto. Debes aplicar un enfoque de ingeniería de software.

¿Cuál es el producto obtenido? Desde el punto de vista de un ingeniero de software, el producto obtenido son los programas, documentos y los datos que configuran el software de computadora. Pero desde el punto de vista de los usuarios el producto obtenido es la información resultante que hace de algún modo el mundo mejor a los usuarios.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Lee el resto de este libro, selecciona aquellas ideas que son aplicables al software que construyes y aplícalas a tu trabajo.

Cinco años después de que la cuarta edición de este libro fue escrita, el papel del software como «alma mater» ha llegado a ser más obvio. Un director de software de Intemet ha producido su propia economía de 500 billones de Euros. En la euforia creada por la promesa de un paradigma económico nuevo, los inversores de Wall Street dieron a las pequeñas empresas «punto-com» estimaciones en billones de dólares antes de que éstas comenzasen a producir un dólar en ventas. Han surgido nuevas industrias dirigidas por software y las antiguas que no se han adaptado a esta nueva tendencia están ahora amenazadas de extinción. El gobierno de Estados Unidos ha mantenido un contencioso frente a la mayor compañía de la industria del software, como lo mantuvo hace poco tiempo cuando se movilizó para detener las actividades monopolísticas en las industrias del acero y del aceite.

El impacto del software en nuestra sociedad y en la cultura continúa siendo profundo. Al mismo tiempo que crece su importancia, la comunidad del software trata continuamente de desarrollar tecnologías que hagan más sencillo, rápido **y** menos costosa la construcción de programas de computadora de alta calidad.

Este libro presenta un marco de trabajo que puede ser usado por aquellos que construyen software informático —aquellos que lo deben hacer bien—. La tecnología que comprende un proceso, un juego de métodos y un conjunto de herramientas **se** llama *ingeniería **del** software.*

1.1. LA EVOLUCIÓN DEL SOFTWARE

Hoy en día el software tiene un doble papel. Es un producto y, al mismo tiempo, el vehículo para entregarlo. Como producto, hace entrega de la potencia informática que incorpora el hardware informático o, más ampliamente, una red de computadoras que es accesible por hardware local. Si reside dentro de un teléfono celular u opera dentro de una computadora central, el software es un transformador de información, produciendo, gestionando, adquiriendo, modificando, mostrando o transmitiendo información que puede ser tan simple como un solo bit, o tan complejo como una presentación en multimedia. Como vehículo utilizado para hacer entrega del producto, el software actúa como la base de control de la computadora (sistemas operativos), la comunicación de información (redes) y la creación y control de otros programas (herramientas de software y entomos).



El software es tonto un producto, como el vehículo poro su entrego

El papel del software informático ha sufrido un cambio significativo durante un periodo de tiempo superior a 50 años. Enormes mejoras en rendimiento del hardware, profundos cambios de arquitecturas informáticas, grandes aumentos de memoria y capacidad de almacenamiento y una gran variedad de opciones de entrada y salida han conducido a sistemas más sofisticados y más complejos basados en computadora. La sofisticación y la complejidad pueden producir resultados deslumbrantes cuando un sistema tiene éxito, pero también pueden suponer grandes problemas para aquellos que deben construir sistemas complejos.

Libros populares publicados durante los años 70 **y** 80 proporcionan una visión histórica útil dentro de la percepción cambiante de las computadoras y del software, **y** de **su** impacto en nuestra cultura. Osborne [OSB79] hablaba de una «nueva revolución industrial». Toffler [TOF80] llamó a la llegada de componentes microelectrónicos la «tercera ola del cambio» en la historia de la humanidad, **y** Naisbitt [NAI82] predijo la transformación de la sociedad industrial a una «sociedad de información». Feigenbaum y McCorduck [FEI83] sugirieron que la información **y** el conocimiento (controlados por computadora) serían el foco de poder del siglo veintiuno, y Stoll [STO89] argumentó que la «comunidad electrónica» creada mediante redes **y** software es la clave para el intercambio de conocimiento alrededor del mundo.

Al comienzo de los años 90, Toffler [TOF90] describió un «cambio de poder» en el que las viejas estructuras de poder (gubernamentales, educativas, industriales, económicas **y** militares) se desintegrarían a medida que



Me introduje en el futuro, más allá de lo que el ojo humano puede ver. Tuve una visión del mundo y todo lo maravilloso que podría ser.
Tennyson

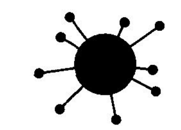
las computadoras y el software nos llevaran a la «democratización del conocimiento». A Yourdon [YOU92] le preocupaba que las compañías en Estados Unidos pudieran perder **su** competitividad en empresas relativas al software y predijo «el declive y la caída del programador americano». Hammer y Champy [HAM93] argumentaron que las tecnologías de información iban a desempeñar el papel principal en la «reingeniería de la compañía». A mediados de los años 90, la persistencia de las computadoras y del software generó una erupción de libros por «neo-Luddites» (por ejemplo: *Resisting the Virtual Life,* editado por James Brook y Ian Boal, y *The Future Does not Compute* de Stephen Talbot). Estos autores critican enormemente la computadora, haciendo énfasis en preocupaciones legítimas pero ignorando los profundos beneficios que se han llevado a cabo [LEV95].



Los computadoras hacen los cosas más fáciles, pero la mayoría de los cosas que facilitan no es preciso hacerlas.
Andy Rooney

Al final de los años 90, Yourdon [YOU96] volvió a evaluar las perspectivas del software profesional y sugirió la «resurrección y elevación» del programador americano. A medida que internet creció en importancia, **su** cambio de pensamiento demostró ser correcto. Al final del siglo veinte, el enfoque cambió una vez más. Aquí tuvo lugar el impacto de la «bomba de relojería» Y2K (por ejemplo: [YOU98b], [DEJ98], [KAR99]). Aunque muchos vieron las predicciones de los críticos del Y2K como reacciones, sus populares lecturas devolvieron la difusión del software a sus vidas. Hoy en día, «la computación omnipresente» [NOR98] ha producido una generación de aplicaciones de información que tienen conexión en banda ancha a la Web para proporcionar «una capa de conexión sobre nuestras casas, oficinas, y autopistas» [LEV99]. El papel del software continúa su expansión.

El programador solitario de antaño ha sido reemplazado por un equipo de especialistas del software, cada uno centrado en una parte de la tecnología requerida para entregar una aplicación concreta. Y de este modo, las cuestiones que se preguntaba el programador solitario son las mismas cuestiones que nos preguntamos cuando construimos sistemas modernos basados en computadoras:



Estadísticas globales de software

- ¿Por qué lleva tanto tiempo terminar los programas?
- ¿Por qué son tan elevados los costes de desarrollo?
- ¿Por qué no podemos encontrar todos los errores antes de entregar el software a nuestros clientes?
- ¿Por qué nos resulta difícil constatar el progreso conforme se desarrolla el software?

1.2 EL SOFTWARE

En 1970, menos del uno por ciento de las personas podría haber descrito inteligentemente lo que significaba «software de computadora». Hoy, la mayoría de los profesionales y muchas personas en general piensan en su mayoría que comprenden el software. ¿Pero lo entienden realmente?

1.2.1. Características del software

Para poder comprender lo que es el software (y consecuentemente la ingeniería del software), es importante examinar las características del software que lo diferencian de otras cosas que los hombres pueden construir. Cuando se construye hardware, el proceso creativo humano (análisis, diseño, construcción, prueba) se traduce finalmente en una forma física. Si construimos una nueva computadora, nuestro boceto inicial, diagramas formales de diseño y prototipo de prueba, evolucionan hacia un producto físico (chips, tarjetas de circuitos impresos, fuentes de potencia, etc.).

El software es un elemento del sistema que es lógico, en lugar de físico. Por tanto el software tiene unas características considerablemente distintas a las del hardware:



El software se desarrolla, no se fabrica.

1. El software se desarrolla, no se fabrica en un sentido clásico.

Aunque existen similitudes entre el desarrollo del software y la construcción del hardware, ambas actividades son fundamentalmente diferentes. En ambas actividades la buena calidad se adquiere mediante un buen diseño, pero la fase de construcción del hardware puede introducir problemas de calidad que no existen (o son fácilmente corregibles) en el software. Ambas actividades dependen de las personas, pero la relación entre las personas dedicadas y el trabajo realizado es completamente diferente para el software (véase el Capítulo 7). Ambas actividades requieren la construcción de un «producto» pero los enfoques son diferentes.

Los costes del software se encuentran en la ingeniería. Esto significa que los proyectos de software no se pueden gestionar como si fueran proyectos de fabricación.

2. *El software no se «estropea».*

La Figura 1.1 describe, para el hardware, la proporción de fallos como una función del tiempo. Esa relación, denominada frecuentemente «curva de bañera», indica que el hardware exhibe relativamente muchos fallos al principio de su vida (estos fallos son atribuibles normalmente a defectos del diseño o de la fabricación); una vez corregidos los defectos, la tasa de fallos cae hasta un nivel estacionario (bastante bajo, con un poco de optimismo) donde permanece durante un cierto periodo de tiempo. Sin embargo, conforme pasa el tiempo, el hardware empieza a desgastarse y la tasa de fallos se incrementa.

El software no es susceptible a los males del entorno que hacen que el hardware se estropee. Por tanto, en teoría, la curva de fallos para el software tendría la forma que muestra la Figura **1.2.** Los defectos no detectados haran que falle el programa durante las primeras etapas de su vida. Sin embargo, una vez que se corrigen (suponiendo que no se introducen nuevos errores) la curva se aplana, como se muestra. La curva idealizada es una gran simplificación de los modelos reales de fallos del software (vease más información en el Capítulo **8).** Sin embargo la implicación es clara, el software no se estropea. ¡Pero se deteriora!



El software no se estropea, pero se deteriora.

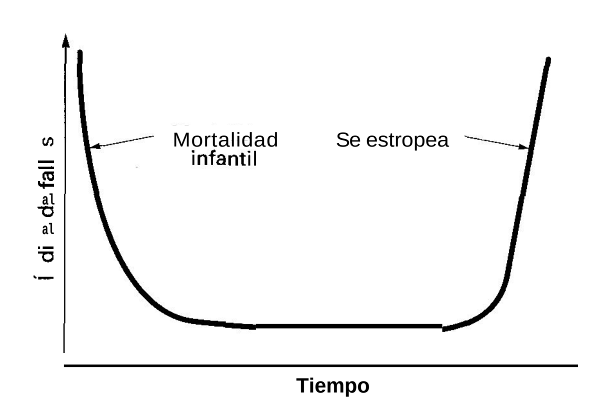


FIGURA 1.1. Curva de fallos del hardware.

Esto que parece una contradicción, puede comprenderse mejor considerando «la curva actual» mostrada en la Figura 1.2. Durante su vida, el software sufre cambios (mantenimiento). Conforme se hacen los cambios, es bastante probable que se introduzcan nuevos defectos, haciendo que la curva de fallos tenga picos como se ve en la Figura 1.2. Antes de que la curva pueda volver al estado estacionario original, se solicita otro cambio, haciendo que de nuevo se cree otro pico. Lentamente, el nivel mínimo de fallos comienza a crecer —el software se va deteriorando debido a los cambios—.

Otro aspecto de ese deterioro ilustra la diferencia entre el hardware y el software. Cuando un componente de hardware se estropea se sustituye por una pieza de repuesto. No hay piezas de repuesto para el software. Cada fallo en el software indica un error en el diseño o en el proceso mediante el que se tradujo el diseño a código máquina ejecutable. Por tanto, el mantenimiento del software tiene una complejidad considerablemente mayor que la del mantenimiento del hardware.

3. *Aunque la industria tiende a ensamblar componentes, la mayoría del software se construye a medida.*

Consideremos la forma en la que se diseña y se construye el hardware de control para un producto basado en computadora. El ingeniero de diseño construye un sencillo esquema de la circuitería digital, hace algún análisis fundamental para asegurar que se consigue la función adecuada y va al armario donde se encuentran los catálogos de componentes digitales. Después de seleccionar cada componente, puede solicitarse la compra.

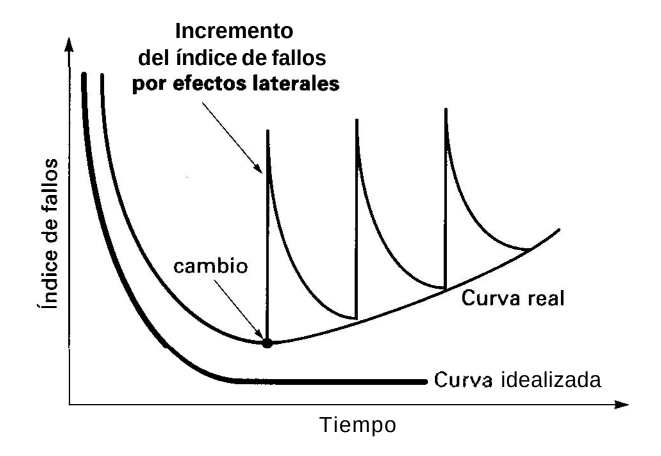


FIGURA 1.2. Curvas de fallos real e idealizada del software.



CLAVE

Los métodos de ingeniería de software se esfuerzan para reducir la magnitud de los picos y la inclinación de la curva (Fig. 1.2).

A medida que la disciplina del software evoluciona, se crea un grupo de componentes de diseño estándar. Tornillos estándar y circuitos integrados preparados para la venta son solamente los dos mil coinponentes estándar que utilizan ingenieros mecánicos y eléctricos cuando diseñan nuevos sistemas. Los componentes reutilizables se han creado para que el ingeniero pueda concentrarse en elementos verdaderamente innovadores de un diseño, por ejemplo, las partes del diseño que representan algo nuevo. En el mundo del hardware, la reutilización de componentes es una parte natural del proceso de ingeniería. En el mundo del software es algo que sólo ha comenzado a lograrse en una escala amplia.

El componente de software debería diseñarse e implementarse para que pueda volver a ser reutilizado en muchos programas diferentes. En los años 60, se construyeron bibliotecas de subrutinas científicas reutilizables en una amplia serie de aplicaciones científicas y de ingeniería. Esas bibliotecas de subrutinas reutilizaban de forma efectiva algoritmos bien definidos, pero tenían un dominio de aplicación limitado. Hoy en día, hemos extendido nuestra visión de reutilización para abarcar no sólo los algorítmos, sino también estructuras de datos. Los componentes reutilizables modernos encapsulan tanto datos como procesos que se aplican a los datos, permitiendo al ingeniero del software crear nuevas aplicaciones a partir de las partes reutilizables. Por ejemplo, las interfaces gráficas de usuario de hoy en día se construyen frecuentemente a partir de componentes reutilizables que permiten la creación de ventanas gráficas, de menús despleglables y de una amplia variedad de mecanismos de interacción.

C VE

La mayoría del software sigue construyéndose a medida.

1.2.2. Aplicaciones del software

El software puede aplicarse en cualquier situación en la que se haya definido previamente un conjunto específico de pasos procedimentales (es decir, un algoritmo) (excepciones notables a esta regla son el software de los sistemas expertos y de redes neuronales). El contenido y el determinismo de la información son factores importantes a considerar para determinar la naturaleza de una aplicación de software. El contenido se refiere al significado y a la forma de la información de entrada y salida. Por ejemplo, muchas aplicaciones bancarias usan unos datos de entrada muy estructurados (una base de datos) y producen «informes» con determinados formatos. El software que controla una máquina automática (por ejemplo: un control numérico) acepta elementos de datos discretos con una estructura limitada y produce órdenes concretas para la máquina en rápida sucesión.

Referencia cruzada

l a revolución del software se foto en el Capítulo 13. Lo ingeniería de software basada en componentes se presento en el Capítulo 27.

El determinismo de la información se refiere a la predecibilidad del orden y del tiempo de llegada de los datos. Un programa de análisis de ingeniería acepta datos que están en un orden predefinido, ejecuta el algoritmo(s) de análisis sin interrupción y produce los datos resultantes en un informe o formato gráfico. Se dice que tales aplicaciones son determinadas. Un sistema operativo multiusuario, por otra parte, acepta entradas que tienen un contenido variado y que se producen en instantes arbitrarios, ejecuta algoritmos que pueden ser interrumpidos por condiciones externas y produce una salida que depende de una función del entorno y del tiempo. Las aplicaciones con estas características se dice que son indeterminadas.

Algunas veces es difícil establecer categorías genéricas para las aplicaciones del software que sean significativas. Conforme aumenta la complejidad del software, es más difícil establecer compartimentos nítidamente separados. Las siguientes áreas del software indican la amplitud de las aplicaciones potenciales:

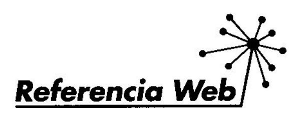
Software de sistemas. El software de sistemas es un conjunto de programas que han sido escritos para servir a otros programas. Algunos programas de sistemas (por ejemplo: compiladores, editores y utilidades de gestión de archivos) procesan estructuras de información complejas pero determinadas. Otras aplicaciones de sistemas (por ejemplo: ciertos componentes del sistema operativo, utilidades de manejo de periféricos, procesadores de telecomunicaciones) procesan datos en gran medida indeterminados. En cualquier caso, el área del software de sistemas se caracteriza por una fuerte interacción con el hardware de la computadora; una gran utilización por múltiples usuarios; una operación concurrente que requiere una planificación, una compartición de recursos y una sofisticada gestión de procesos; unas estructuras de datos complejas y múltiples interfaces externas.

Software de tiempo real. El software que coordina/analiza/controla sucesos del mundo real conforme ocurren, se denomina de tiempo real. Entre los elementos del software de tiempo real se incluyen: un componente de adquisición de datos que recolecta y da formato a la información recibida del entorno externo, un componente de análisis que transforma la información según lo requiera la aplicación, un componente de control/salida que responda al entorno externo, y un componente de monitorización que coordina todos los demás componentes, de forma que pueda mantenerse la repuesta en tiempo real (típicamente en el rango de un milisegundo a un segundo).

Software de gestión. El proceso de la información comercial constituye la mayor de las áreas de aplicación del software. Los «sistemas» discretos (por ejemplo: nóminas, cuentas de haberes-débitos, inventarios, etc.) han evolucionado hacia el software de sistemas de información de gestión **(SIG)** que accede a una o más bases de datos que contienen información comercial. Las aplicaciones en esta área reestructuran los datos existentes para facilitar las operaciones comerciales o gestionar la toma de decisiones. Además de las tareas convencionales de procesamientos de datos, las aplicaciones de software de gestión también realizan cálculo interactivo (por ejemplo: el procesamiento de transacciones en puntos de ventas).

Software de ingeniería y científíco. El software de ingeniería y científico está caracterizado por los algoritmos de «manejo de números». Las aplicaciones van desde la astronomía a la vulcanología, desde el análisis de la presión de los automotores a la dinámica orbital de las lanzaderas espaciales y desde la biología molecular a la fabricación automática. Sin embargo, las nuevas aplicaciones del área de ingeniería/ciencia se han alejado de los algoritmos convencionales numéricos. El diseño asistido por computadora (del inglés CAD), la simulación de sistemas y otras aplicaciones interactivas, han comenzado a coger características del software de tiempo real e incluso del software de sistemas.

Software empotrado. Los productos inteligentes se han convertido en algo común en casi todos los mercados de consumo e industriales. El software empotrado reside en memoria de sólo lectura y se utiliza para controlar productos y sistemas de los mercados industriales y de consumo. El software empotrado puede ejecutar funciones muy limitadas y curiosas (por ejemplo: el control de las teclas de un horno de microondas) o suministrar una función significativa y con capacidad de control (por ejemplo: funciones digitales en un automóvil, tales como control de la gasolina, indicadores en el salpicadero, sistemas de frenado, etc.).



Referencia Web

Se puede encontrar una de las mayores bibliotecas de *shareware/freeware* en **www.shareware.com**

Software de computadoras personales. El mercado del software de computadoras personales ha germinado en las pasadas dos décadas. El procesamiento de textos, las hojas de cálculo, los gráficos por computadora, multimedia, entretenimientos, gestión de bases de datos, aplicaciones financieras, de negocios y personales y redes o acceso a bases de datos externas son algunas de los cientos de aplicaciones.

Software basado en Web. Las páginas Web buscadas por un explorador son software que incorpora instrucciones ejecutables (por ejemplo, CGI, HTML, Perl, o Java), y datos (por ejemplo, hipertexto y una variedad de formatos de audio y visuales). En esencia, la red viene a ser una gran computadora que proporciona un recurso software casi ilimitado que puede ser accedido por cualquiera con un modem.

Software de inteligencia artificial. El software de inteligencia artificial (IA) hace uso de algoritmos no numéricos para resolver problemas complejos para los que no son adecuados el cálculo o el análisis directo. Los sistemas expertos, también llamados sistemas basados en el conocimiento, reconocimiento de patrones (imágenes y voz), redes neuronales artificiales, prueba de teoremas, y los juegos son representativos de las aplicaciones de esta categoría.

1.3 SOFTWARE: ¿UNA CRISIS EN EL HORIZONTE?

Muchos observadores de la industria (incluyendo este autor) han caracterizado los problemas asociados con el desarrollo del software como una «crisis». Más de unos cuantos libros (por ejemplo: [GLA97], [FLO97], [YOU98a]) han recogido el impacto de algunos de los fallos mas importantes que ocurrieron durante la década pasada. No obstante, los mayores éxitos conseguidos por la industria del software han llevado a preguntarse si el término (crisis del software) es aún apropiado. Robert Glass, autor de varios libros sobre fallos del software, representa a aquellos que han sufrido un cambio de pensamiento. Expone [GLA98]: «Puedo ver en mis ensayos históricos de fallos y en mis informes de excepción, fallos importantes en medio de muchos éxitos, una copa que está [ahora] prácticamente llena.»

Cita:
La mayoría de los expertos están de acuerdo en que la manera más probable para que el mundo se destruya es por accidente. Ahí es donde nosotros entramos; somos profesionales de la informática, provocamos accidentes.
Nathaniel Borenstein

La palabra crisis se define en el *diccionario Webster* como «un punto decisivo en el curso de algo, momento, etapa o evento decisivo o crucial». Sin embargo, en términos de calidad del software total y de velocidad con la cual son desarrollados los productos y los sistemas basados en computadoras, no ha habido ningún «punto crucial», ningún «momento decisivo», solamente un lento cambio evolutivo, puntualizado por cambios tecnológicos explosivos en las disciplinas relacionadas con el software.

Cualquiera que busque la palabra *crisis* en el diccionario encontrará otra definición: «el punto decisivo en el curso de una enfermedad, cuando se ve más claro si el paciente vivirá o morirá». Esta definición puede darnos una pista sobre la verdadera naturaleza de los problemas que han acosado el desarrollo del software.

Lo que realmente tenemos es una aflicción crónica[1]. La palabra *aflicción* se define como «algo que causa pena o desastre». Pero la clave de nuestro argumento es la definición del adjetivo *crónica:* «muy duradero *o* que reaparece con frecuencia continuando indefinidamente». Es bastante más preciso describir los problemas que hemos estado aguantando en el negocio del software como una aflicción crónica, en vez de como una crisis.

Sin tener en cuenta como lo llamemos, el conjunto de problemas encontrados en el desarrollo del software de computadoras no se limitan al software que «no funciona correctamente». Es más, el mal abarca los problemas asociados a cómo desarrollar software, cómo mantener el volumen cada vez mayor de software existente y cómo poder esperar mantenemos al corriente de la demanda creciente de software.

Vivimos con esta aflicción desde este día —de hecho, la industria prospera a pesar de ello—. Y así, las cosas podrán ser mejores si podemos encontrar y aplicar un remedio.

1.4 MITOS DEL SOFTWARE

Muchas de las causas de la crisis del software se pueden encontrar en una mitología que surge durante los primeros años del desarrollo del software. A diferencia de los mitos antiguos, que a menudo proporcionaban a los hombres lecciones dignas de tener en cuenta, los mitos del software propagaron información errónea y confusión. Los mitos del software tienen varios atributos que los hacen insidiosos: por ejemplo, aparecieron como declaraciones razonables de hechos (algunas veces conteniendo elementos verdaderos), tuvieron un sentido intuitivo y frecuentemente fueron promulgados por expertos que «estaban al día».

[1] Esta terminología fue sugerida por el profesor Daniel Tiechrow de la Universidad de Michigan en una conferencia impartida en Ginebra, Suiza, Abril, 1989.

Mitos de gestión. Los gestores con responsabilidad sobre el software, como los gestores en la mayoría de las disciplinas, están normalmente bajo la presión de cumplir los presupuestos, hacer que no se retrase el proyecto y mejorar la calidad. Igual que se agarra al vacío una persona que se ahoga, un gestor de software se agarra frecuentemente a un mito del software, aunque tal creencia sólo disminuya la presión temporalmente.

Mito. Tenemos ya un libro que está lleno de estándares y procedimientos para construir software, ¿no le proporciona ya a mi gente todo lo que necesita saber?

Realidad. Está muy bien que el libro exista, pero ¿se usa?, ¿conocen los trabajadores su existencia?, ¿refleja las prácticas modernas de desarrollo de software?, ¿es completo?, ¿está diseñado para mejorar el tiempo de entrega mientras mantiene un enfoque de calidad? En muchos casos, la respuesta a todas estas preguntas es «no».

Cita:
En ausencia de estándares significativos, una nueva industria como la del software pasa a depender de las costumbres.
Tom De Marco

Mito. Mi gente dispone de las herramientas de desarrollo de software más avanzadas, después de todo, les compramos las computadoras más modernas.

Realidad. Se necesita mucho más que el último modelo de computadora grande o de PC para hacer desarrollo de software de gran calidad. Las herramientas de ingeniería del software asistida por computadora (CASE) son más importantes que el hardware para conseguir buena calidad y productividad, aunque la mayoría de los desarrolladores del software todavía no las utilicen eficazmente.

Mito. Si fallamos en la planificación, podemos añadir más programadores y adelantar el tiempo perdido (el llamado algunas veces «concepto de la horda Mongoliana»).

Realidad. El desarrollo de software no es un proceso mecánico como la fabricación. En palabras de Brooks [BRO75]: «...añadir gente a un proyecto de software retrasado lo retrasa aún más». Al principio, esta declaración puede parecer un contrasentido. Sin embargo, cuando se añaden nuevas personas, la necesidad de aprender y comunicarse con el equipo puede y hace que se reduzca la cantidad de tiempo gastado en el desarrollo productivo. Puede añadirse gente, pero sólo de una manera planificada y bien coordinada.



La red de gestión de proyectos de software en **www.spmn.com** puede ayudarle a desmitificar estos y otros mitos.

Mitos del Cliente. Un cliente que solicita una aplicación de software puede ser una persona del despacho de al lado, un grupo técnico de la sala de abajo, el departamento de ventas o una compañía exterior que solicita un software bajo contrato. En muchos casos, el cliente cree en los mitos que existen sobre el software, debido a que los gestores y desarrolladores del software hacen muy poco para corregir la mala información. Los mitos conducen a que el cliente se cree una falsa expectativa y, finalmente, quede insatisfecho con el que desarrolla el software.

Mito. Una declaración general de los objetivos es suficiente para comenzar a escribir los programas —podemos dar los detalles más adelante—.

Realidad. Una mala definición inicial es la principal causa del trabajo baldío en software. Es esencial una descripción formal y detallada del ámbito de la información, funciones, comportamiento, rendimiento, interfaces, ligaduras del diseño y criterios de validación. Estas características pueden determinarse sólo después de una exhaustiva comunicación entre el cliente y el analista.

Mito. Los requisitos del proyecto cambian continuamente, pero los cambios pueden acomodarse fácilmente, ya que el software es flexible.

Referencia cruzada
La gestión y control de cambio está tratada con detalle en el Capítulo 9

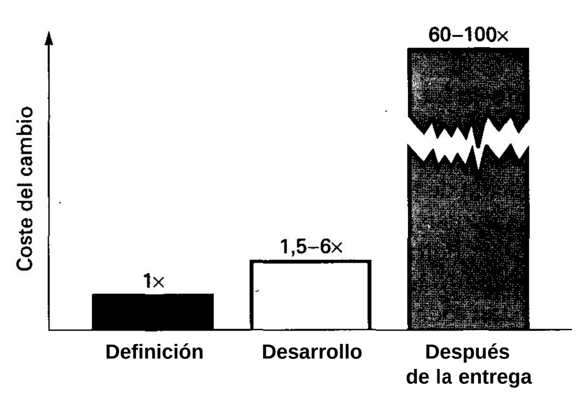


FIGURA 1.3. El impacto del cambio.

Realidad. Es verdad que los requisitos del software cambian, pero el impacto del cambio varía según el momento en que se introduzca. La Figura **1.3** ilustra el impacto de los cambios. Si se pone cuidado al dar la definición inicial, los cambios solicitados al principio pueden acomodarse fácilmente. El cliente puede revisar los requisitos y recomendar las modificaciones con relativamente poco impacto en el coste. Cuando los cambios se solicitan durante el diseño del software, el impacto en el coste crece rápidamente. Ya se han acordado los recursos a utilizar y se ha establecido un marco de trabajo del diseño. Los cambios pueden producir trastornos que requieran recursos adicionales e importantes modificaciones del diseño; es decir, coste adicional. Los

cambios en la función, rendimiento, interfaces u otras características, durante la implementación (codificación y prueba) pueden tener un impacto importante sobre el coste. Cuando se solicitan al final de un proyecto, los cambios pueden producir un orden de magnitud más caro que el mismo cambio pedido al principio.

Mitos de los desarrolladores. Los mitos en los que aún creen muchos desarrolladores se han ido fomentando durante 50 años de cultura informática. Durante los primeros días del desarrollo del software, la programación se veía como un arte. Las viejas formas y actitudes tardan en morir.

Mito. Una vez que escribimos el programa y hacemos que funcione, nuestro trabajo ha terminado.

Realidad. Alguien dijo una vez: «cuanto más pronto se comience a escribir código, más se tardará en terminarlo». Los datos industriales [LIE80, JON91, PUT97] indican que entre el 60 y el 80 por ciento de todo el esfuerzo dedicado a un programa se realizará después de que se le haya entregado al cliente por primera vez.



Trabaja muy duro poro entender lo que tienes que hacer antes de empezar. No serías capaz de desarrollar codo detalle; por más que sepas, toma el menor riesgo.

Mito. Hasta que no tengo el programa «ejecutándose», realmente no tengo forma de comprobar su calidad.

Realidad. Desde el principio del proyecto se puede aplicar uno de los mecanismos más efectivos para garantizar la calidad del software: *la revisión técnica formal.* La revisión del software (descrito en el Capítulo 8) es un «filtro de calidad» que se ha comprobado que es más efectivo que la prueba, para encontrar ciertas clases de defectos en el software.

Mito. Lo único que se entrega al terminar el proyecto es el programa funcionando.

Realidad. Un programa que funciona es sólo una parte de una *configuración del software* que incluye muchos elementos. La documentación proporciona el fundamento para un buen desarrollo y, lo que es más importante, proporciona guías para la tarea de mantenimiento del software.

Muchos profesionales del software reconocen la falacia de los mitos descritos anteriormente. Lamentablemente, las actitudes y métodos habituales fomentan una pobre gestión y malas prácticas técnicas, incluso cuando la realidad dicta un método mejor. El reconocimiento de las realidades del software es el primer paso hacia la formulación de soluciones prácticas para su desarrollo.

RESUMEN

El software se ha convertido en el elemento clave de la evolución de los sistemas y productos informáticos. En los pasados 50 años, el software ha pasado de ser una resolución de problemas especializada y una herramienta de análisis de información, a ser una industria por sí misma. Pero la temprana cultura e historia de la «programación» ha creado un conjunto de problemas que persisten todavía hoy. El software se ha convertido en un factor que limita la evolución de los sistemas informáticos. El software se compone de programas, datos y documentos. Cada uno de estos elementos componen una configuración que se crea como parte del proceso de la ingeniería del software. El intento de la ingeniería del software es proporcionar un marco de trabajo para construir software con mayor calidad.



Cuando te pones o pensar, no encuentras tiempo poro la disciplino de lo ingeniería del software, y te preguntas: «¿Tendré tiempo para poder hacerlo?»

REFERENCIAS

[BRO75] Brooks, F., *The Mytical Man-Month,* Addison-Wesley, 1975.

[DEJ98] De Jager, P., et al, *Countdown Y2K: Business Survival Planning for the Year 2000,* Wiley, 1998.

[DEM95] De Marco, T., *Why Does Software Cost So Much?,* Dorset House, 1995, p. 9.

[FEI83] Feigenbaum, E. A., y P. McCorduck, *The Fith Generation,* Addison-Wesley, 1983.

[FLO97] Flowers, S., *Software Failure, Management Failure-Amaicing Stories and Cautionary Tails,* Wiley, 1997 (?).

[GLA97] Glass, R. L., *Software Runaways,* Prentice Hall, 1997.

[GLA98] Glass, R. L.,«Is there Really a Software Crisis?», *IEEE Software,* vol. 15, n.º 1, Enero 1998, pp. 104-105.

[HAM93] Hammer, M., y J. Champy, *Reengineering the Corporation,* HarpperCollins Publisher, 1993.

[JON91] Jones, C., *Applied Software Measurement,* McGraw-Hill, 1991.

[KAR99] Karlson, E., y J. Kolber, *A Basic Introduction to Y2K: How the Year 2000 Computer Crisis Affects You?,* Next Era Publications, Inc., 1999.

[LEV95] Levy, S., «The Luddites Are Pack», *Newsweek,* 12 de Julio de 1995, p. 55.

[LEV99] Levy, S., «The New Digital Galaxy», *Newsweek,* 31 de Mayo de 1999, p.57.

[LIE80] Lientz, B., y E. Swanson, *Software Maintenance Management,* Addison Wesley, 1980.

[NAI82] Naisbitt, J., *Megatoends,* Warner Books, 1982.

[NOR98] Norman, D., *The Invisible Computer,* MIT Press, 1998.

[OSB79] Osborne, A., *Running Wild-The Next Industrial Revolution,* Osborne/McGraw-Hill, 1979.

[PUT97] Putnam, L., y W. Myers, *Industrial Strength Software,* IEEE Computer Society Press, 1997.

[STO89] Stoll, C., *The cuckoo's Egg,* Doubleday, 1989.

[TOF80] Toffler, A., *The Third Wave,* Morrow Publishers, 1980.

[TOF90] Toffler, A., *Powershift,* Bantam Publishers, 1990.

[YOU92] Yourdon, E., *The Decline and Fall of the American Programmer,* Yourdon Press, 1992.

[YOU96] Yourdon, E., *The Rise and Resurrection of the American Programmer,* Yourdon Press, 1996.

[YOU98a] Yourdon, E., *Death March Projects,* Prentice-Hall, 1998.

[YOU98b] Yourdon, E., y J. Yourdon, *Time Bomb* 2000, Prentice-Hall, 1998.

PROBLEMAS Y PUNTOS A CONSIDERAR

1.1. El software es la característica que diferencia a muchos productos y sistemas informáticos. Dé ejemplos de dos o tres productos y de, al menos, un sistema en el que el software, no el hardware, sea el elemento diferenciador.

1.2. En los años cincuenta y sesenta la programación de computadoras era un arte aprendido en un entorno básicamente experimental. ¿Cómo ha afectado esto a las prácticas de desarrollo del software hoy?

1.3. Muchos autores han tratado el impacto de la «era de la información». Dé varios ejemplos (positivos y negativos) que indiquen el impacto del software en nuestra sociedad. Repase algunas referencias de la Sección 1.1 previas a 1990 e indique dónde las predicciones del autor fueron correctas y dónde no lo fueron.

1.4. Seleccione una aplicación específica e indique: (a) la categoría de la aplicación de software (Sección 1.2.2) en la que encaje; (b) el contenido de los datos asociados con la aplicación; (c) la información determinada de la aplicación.

1.5. A medida que el software se difunde más, los riesgos para el público (debido a programas defectuosos) se convierten en una preocupación cada vez más significativa. Desarrolle un escenario realista del juicio final (distinto a Y2K) en donde el fallo de computadora podría hacer un gran daño (económico o humano).

1.6. Lea detenidamente el grupo de noticias de Internet **comp.risk** y prepare un resumen de riesgos para las personas con las que se hayan tratado Últimamente. Código alternativo: *Software Engineering Notes* publicado por la ACM.

1.7. Escriba un papel que resuma las ventajas recientes en una de las áreas de aplicaciones de software principales. Entre las selecciones potenciales se incluyen: aplicaciones avanzadas basadas en Web, realidad virtual, redes neuronales artificiales, interfaces humanas avanzadas y agentes inteligentes.

1.8. Los mitos destacados en la Sección 1.4 se están viniendo abajo lentamente a medida que pasan los años. Pero otros se están haciendo un lugar. Intente añadir un mito o dos mitos «nuevos» a cada categoría.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Literalmente existen miles de libros escritos sobre software de computadora. La gran mayoría tratan los lenguajes de programación o aplicaciones de software, y sólo unos pocos tratan el software en sí. Pressman y Herron *(Software Sock,* Dorset House, 1991) presentaron una discusión (dirigida a no profesionales) acerca del software y del modo en que lo construyen los profesionales.

El libro, éxito de ventas, de Negroponte *(Being Digital,* Alfred A. Knopf, Inc., 1995) proporciona una visión de las computadoras y de su impacto global en el siglo XXI. Los libros de Norman [NOR98] y Bergman *(Information Appliances & Beyond,* Academic Pres/Morgan Kauffman, 2000) sugieren que el impacto extendido del PC declinará al mismo tiempo que las aplicaciones de información y la difusión de la programación conecten a todos en el mundo industrializado y casi todas las aplicaciones a la nueva infraestructura de Internet.

Minasi *(The Software Conspiracy: Why Software Companies Put Out Faulty Products, How They Can Hurt* ***You,*** *and What You Can Do,* McGraw-Hill, 2000) argumentó que el «azote moderno» de los errores del software puede eliminarse y sugiere formas para hacerlo. DeMarco *(Why Does Software Cost So Much?,* Dorset House, 1995) ha producido una colección de ensayos divertidos e interesantes sobre el software y el proceso a través del cual se desarrolla.

En Intemet están disponibles una gran variedad de fuentes de información relacionadas con temas de gestión y de software. Se puede encontrar una lista actualizada con referencias a sitios (páginas) web relevantes en **http://www.pressman5.com.**

.

CAPÍTULO 2

EL PROCESO

HOWARD Baetjer, Jr. [BAE98], en un libro fascinante que proporciona un punto de vista economicista del software y de la ingeniería del software, comenta sobre el proceso:

> Como el software, al igual que el capital, es el conocimiento incorporado, y puesto que el conocimiento está inicialmente disperso, el desarrollo del software implícito, latente e incompleto en gran medida, es un proceso social de aprendizaje. El proceso es un diálogo en el que se reúne el conocimiento y se incluye en el software para convertirse en software. El proceso proporciona una interacción entre los usuarios y los diseñadores, entre los usuarios y las herramientas de desarrollo, y entre los diseñadores y las herramientas de desarrollo [tecnología]. Es un proceso interactivo donde la herramienta de desarrollo se usa como medio de comunicación, con cada iteración del diálogo se obtiene mayor conocimiento de las personas involucradas.

Realmente, construir software de computadora es un proceso de aprendizaje iterativo, y el resultado, algo que Baetjer podría llamar «capital del software», es el conjunto del software reunido, depurado y organizado mientras se desarrolla el proceso.

VISTAZO RÁPIDO

¿Qué es? Cuando trabaja para construir un producto o un sistema, es importante seguir una serie de pasos predecibles —un mapa de carreteras que le ayude a obtener el resultado oportuno de calidad—. El mapa de carreteras a seguir es llamado «proceso del software..

¿Quién lo hace? Los ingenieros de software y sus gestores adaptan el proceso a sus necesidades y entonces lo siguen. Además las personas que han solicitado el software tienen un papel a desempeñar en el proceso del software.

¿Por qué es importante? Porque proporciona estabilidad, control y organización a una actividad que puede, si no se controla, volverse caótica.

¿Cuáles son los pasos? A un nivel detallado, el proceso que adoptemos depende del software que estamos construyendo. Un proceso puede ser apropiado para crear software de un sistema de aviación, mientras que un proceso diferente por completo puede ser adecuado para la creación de un sitio web.

¿Cuál es el producto obtenido? Desde el punto de vista de un ingeniero de software, los productos obtenidos son programas, documentos y datos que se producen como consecuencia de las actividades de ingeniería del software definidas por el proceso.

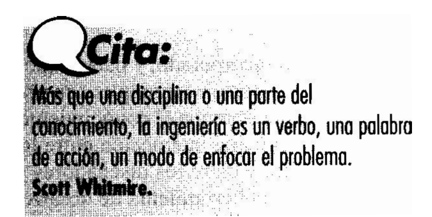
¿Cómo puedo estar seguro de que lo he hecho correctamente? Hay una cantidad de mecanismos de evaluacion del proceso del software que permiten a las organizaciones determinar la «madurez» de su proceso del software. Sin embargo, la calidad, oportunidad y viabilidad a largo plazo del producto que está construyendo son los mejores indicadores de la eficiencia del proceso que estamos utilizando.

Pero, ¿qué es exactamente el proceso del software desde un punto de vista técnico? Dentro del contexto de este libro, definimos un proceso de software como un marco de trabajo de las tareas que se requieren para construir software de alta calidad. ¿Es «proceso» sinónimo de ingeniería del software? La respuesta es «sí» y «no». Un proceso de software define el enfoque que se toma cuando el software es tratado por la ingeniería. Pero la ingeniería del software también comprende las tecnologías que tiene el proceso —métodos técnicos y herramientas automatizadas—.

Aún más importante es que la ingeniería del software la realizan personas creativas, con conocimiento, que deberían trabajar dentro de un proceso del software definido y avanzado que es apropiado para los productos que construyen y para las demandas de su mercado. La intención de este capítulo es proporcionar un estudio del estado actual del proceso del software y puntualizar sobre el estudio detallado de los temas de gestión y técnicos presentados en este libro.

2.1 INGENIERÍA DEL SOFTWARE: UNA TECNOLOGÍA ESTRATIFICADA

Aunque cientos de autores han desarrollado definiciones personales de la ingeniería del software, una definición propuesta por Fritz Bauer [NAU69] en una conferencia de gran influencia sobre estos temas va a servir como base de estudio:



[La ingeniería del software] es el establecimiento y uso de principios robustos de la ingeniería a fin de obtener económicamente software que sea fiable y que funcione eficientemente sobre máquinas reales.

Casi todos los lectores tendrán la tentación de seguir esta definición. No dice mucho sobre los aspectos técnicos de la calidad del software; no se enfrenta directamente con la necesidad de la satisfacción del cliente o de la entrega oportuna del producto; omite la mención de la importancia de mediciones y métricas; tampoco expresa la importancia de un proceso avanzado. Y sin embargo, la definición de Bauer nos proporciona una línea base. ¿Cuáles son los «principios robustos de la ingeniería» aplicables al desarrollo de software de computadora? ¿Cómo construimos el software «económicamente» para que sea «fiable»? ¿Qué se necesita para crear programas de computadora que funcionen «eficientemente» no en una máquina si no en diferentes «máquinas reales»? Éstas son cuestiones que siguen siendo un reto para los ingenieros del software.



El IEEE [IEE93] ha desarrollado una definición más completa:

Ingeniería del software: (1) La aplicación de un enfoque sistemático, disciplinado y cuantificable hacia el desarrollo, operación y mantenimiento del software; es decir, la aplicación de ingeniería al software. (2) El estudio de enfoques como en (1).

2.1.1. Proceso, métodos y herramientas

La Ingeniería del software es un tecnología multicapa. Como muestra la Figura 2.1, cualquier enfoque de ingeniería (incluida ingeniería del software) debe apoyarse sobre un compromiso de organización de calidad.

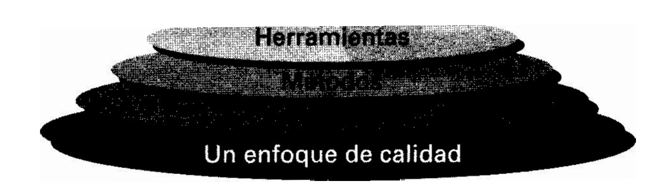


FIGURA 2.1. Capas de la ingeniería del software.

El fundamento de la ingeniería del software es la capa de *proceso.* El proceso de la ingeniería del software es la unión que mantiene juntas las capas de tecnología y que permite un desarrollo racional y oportuno de la ingeniería del software. El proceso define un marco de trabajo para un conjunto de *Úreas clave de proceso* (ACPs) [PAU93] que se deben establecer para la entrega efectiva de la tecnología de la ingeniería del software. Las áreas claves del proceso forman la base del control de gestión de proyectos del software y establecen el contexto en el que se aplican los métodos técnicos, se obtienen productos del trabajo (modelos, documentos, datos, informes, formularios, etc.), se establecen hitos, se asegura la calidad **y** el cambio se gestiona adecuadamente.

Los *métodos* de la ingeniería del software indican «cómo» construir técnicamente el software. Los métodos abarcan una gran gama de tareas que incluyen **aná**lisis de requisitos, diseño, construcción de programas, pruebas y mantenimiento. Los métodos de la ingeniería del software dependen de un conjunto de principios básicos que gobiernan cada área de la tecnología e incluyen actividades de modelado y otras técnicas descriptivas.



CLAVE

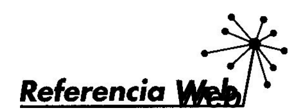
La Ingeniería de software comprende un proceso, métodos técnicos y de gestión, y herramientas.

Las *herramientas* de la Ingeniería del software proporcionan un enfoque automático o semi-automático para el proceso y para los métodos. Cuando se integran herramientas para que la información creada por una herramienta la pueda utilizar otra, se establece un sistema de soporte para el desarrollo del software llamado *ingeniería del software asistida por computadora* ***(CASE).***

2.1.2. Una visión general de la ingeniería del software

La ingeniería es el análisis, diseño, construcción, verificación **y** gestión de entidades técnicas (o sociales). Con independencia de la entidad a la que se va a aplicar ingeniería, se deben cuestionar y responder las siguientes preguntas:

- ¿Cuál es el problema a resolver?
- ¿Cuáles son las características de la entidad que se utiliza para resolver el problema?
- ¿Cómo se realizará la entidad (y la solución)?
- ¿Cómo se construirá la entidad?
- ¿Qué enfoque se va a utilizar para no contemplar los errores que se cometieron en el diseño y en la construcción de la entidad?
- ¿Cómo se apoyará la entidad cuando usuarios soliciten correcciones, adaptaciones y mejoras de la entidad?



Crosstalk es un periódico que proporciona consejos y comentarios prócticos de ingeniería del software. Estón disponibles temas relacionados directamente en: **www.stc.hill.af.mil**

A lo largo de este libro, nos vamos a centrar en una sola entidad —el software de computadora—. Para construir la ingeniería del software adecuadamente, se debe definir un proceso de desarrollo de software. En esta sección se consideran las características genéricas del proceso de software. Más adelante, en este mismo capítulo, se tratarán modelos específicos de procesos.

El trabajo que se asocia a la ingeniería del software se puede dividir en tres fases genéricas, con independencia del área de aplicación, tamaño o complejidad del proyecto. Cada fase se encuentra con una o varias cuestiones de las destacadas anteriormente.

La *fase de definición* se centra sobre el *qué*. Es decir, durante la definición, el que desarrolla el software intenta identificar qué información ha de ser procesada, qué función y rendimiento se desea, qué comportamiento del sistema, qué interfaces van a ser establecidas, qué restricciones de diseño existen, y qué criterios de validación se necesitan para definir un sistema correcto. Por tanto, han de identificarse los requisitos clave del sistema y del software. Aunque los métodos aplicados durante la fase de definición variarán dependiendo del paradigma de ingeniería del software (o combinación de paradigmas) que se aplique, de alguna manera tendrán lugar tres tareas principales: ingeniería de sistemas o de información (Capítulo 10), planificación del proyecto del software (Capítulos 3, 5, 6 y 7) y análisis de los requisitos (Capítulos 11, 12 y 21).



El software se crea aplicando tres fases distintas que se centran en la definición, desarrollo y mantenimiento.

La *fase de desarrollo* se centra en el *cómo*. Es decir, durante el desarrollo un ingeniero del software intenta definir cómo han de diseñarse las estructuras de datos, cómo ha de implementarse la función dentro de una arquitectura de software, cómo han de implementarse los detalles procedimentales, cómo han de caracterizarse interfaces, cómo ha de traducirse el diseño en un lenguaje de programación (o lenguaje no procedimental) y cómo ha de realizarse la prueba. Los métodos aplicados durante la fase de desarrollo variarán, aunque las tres tareas específicas técnicas deberían ocurrir siempre: diseño del software (Capítulos 14, 15 y 21), generación de código y prueba del software (Capítulos 16, 17 y 22).

Cita:

Einstein argumentó que debe haber una explicación simplificada de la naturaleza, porque Dios no es caprichoso ni arbitrario. Esta fe no se ajusta al ingeniero de software.
Gran parte de la complejidad que debe dominar es arbitraria.
Fred Brooks

La *fase de mantenimiento* se centra en el *cambio* que va asociado a la corrección de errores, a las adaptaciones requeridas a medida que evoluciona el entorno del software y a cambios debidos a las mejoras producidas por los requisitos cambiantes del cliente. Durante la fase de mantenimiento se encuentran cuatro tipos de cambios:

Corrección. Incluso llevando a cabo las mejores actividades de garantía de calidad, es muy probable que el cliente descubra los defectos en el software. El *mantenimiento correctivo* cambia el software para corregir los defectos.

Adaptación. Con el paso del tiempo, es probable que cambie el entorno original (por ejemplo: CPU, el sistema operativo, las reglas de empresa, las características externas de productos) para el que se desarrolló el software. El *mantenimiento adaptativo* produce modificación en el software para acomodarlo a los cambios de su entorno externo.

Mejora. Conforme se utilice el software, el cliente/usuario puede descubrir funciones adicionales que van a producir beneficios. El *mantenimiento perfectivo* lleva al software más allá de sus requisitos funcionales originales.

Prevención. El software de computadora se deteriora debido al cambio, y por esto el *mantenimiento preventivo* también llamado *reingeniería del software*, se debe conducir a permitir que el software sirva para las necesidades de los usuarios finales. En esencia, el mantenimiento preventivo hace cambios en programas de computadora a fin de que se puedan corregir, adaptar y mejorar más fácilmente.

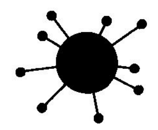
Además de estas actividades de mantenimiento, los usuarios de software requieren un mantenimiento continuo. Los asistentes técnicos a distancia, teléfonos de ayuda y sitios Web de aplicaciones específicas se implementan frecuentemente como parte de la fase de mantenimiento.



Cuando utilizamos el término «mantenimiento» reconocemos que es mucho más que una simple corrección de errores.

Hoy en día, el aumento de los programas[1] legales está forzando a muchas compañías a seguir estrategias de reingeniería del software (Capítulo 30). En un sentido global, la reingeniería del software se considera a menudo como una parte de la reingeniería de procesos comerciales [STA95].

Las fases y los pasos relacionados descritos en nuestra visión genérica de la ingeniería del software se complementan con un número de *actividades protectoras.*



Actividades de protección

Entre las actividades típicas de esta categoría se incluyen:

- Seguimiento y control del proyecto de software
- Revisiones técnicas formales
- Garantía de calidad del software
- Gestión de configuración del software
- Preparación y producción de documentos
- Gestión de reutilización
- Mediciones
- Gestión de riesgos

Las actividades de protección se aplican a lo largo de todo el proceso del software y se tratan en las partes Segunda y Quinta del libro.

2.2 EL PROCESO DEL SOFTWARE

Un proceso de software se puede caracterizar como se muestra en la Figura 2.2. Se establece un *marco común del proceso* definiendo un pequeño número de actividades del marco de trabajo que son aplicables a todos los proyectos del software, con independencia de **su** tamaño o complejidad. Un número de *conjuntos de tareas* - cada uno es una colección de tareas de trabajo de ingeniería del software, hitos de proyectos, productos de trabajo, y puntos de garantía de calidad —que permiten que las actividades del marco de trabajo se adapten a las características del proyecto del software y a los requisitos del equipo del proyecto. Finalmente, las actividades de protección —tales como garantía de calidad del software, gestión de configuración del software y medición* — abarcan el modelo de procesos. Las actividades de protección son independientes de cualquier actividad del marco de trabajo y aparecen durante todo el proceso.

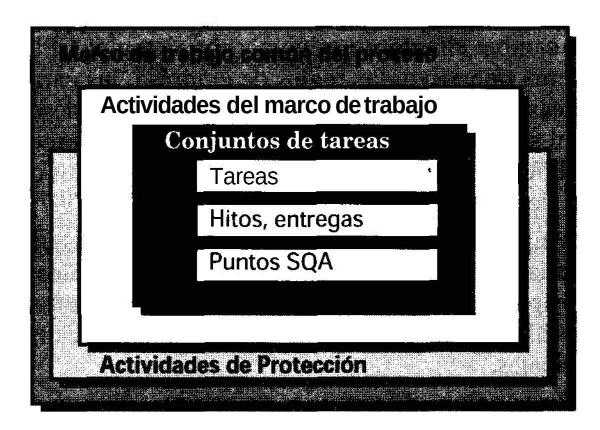


FIGURA 2.2. El proceso del software.

En los Últimos años, se ha hecho mucho énfasis en la «madurez del proceso». El Softwate Engineering Institute (SEI)[3] ha desarrollado un modelo completo que se basa en un conjunto de funciones de ingeniería del software que deberían estar presentes conforme organizaciones alcanzan diferentes niveles de madurez del proceso. Para determinar el estado actual de madurez del proceso de una organización, el SEI utiliza un cuestionario de evaluación y un esquema de cinco grados.



Seleccione un marco de trabajo del proceso común que se adecue al producto, al personal y al proyecto.

El esquema de grados determina la conformidad con un modelo de capacidad de madurez [PAU93] que define las actividades clave que se requieren en los diferentes niveles de madurez del proceso. El enfoque del SEI proporciona una medida de la efectividad global de

[1] El término «programas legales» es un eufemismo para el sottware antiguo, a menudo diseñado y documentado pobremente, que es crítico para el negocio y debe ser soportado durante algunos años.

[2] Estos temas se tratan con más detalle en capítulos posteriores.

[3] El autor se está refiriendo al SEI de la Cannegie Mellon University.

las prácticas de ingeniería del software de una compañía y establece cinco niveles de madurez del proceso, que se definen de la forma siguiente:

Nivel 1: Inicial. El proceso del software se caracteriza según el caso, y ocasionalmente incluso de forma caótica. Se definen pocos procesos, y el éxito depende del esfuerzo individual.

Nivel 2: Repetible. Se establecen los procesos de gestión del proyecto para hacer seguimiento del coste, de la planificación y de la funcionalidad. Para repetir éxitos anteriores en proyectos con aplicaciones similares se aplica la disciplina necesaria para el proceso.

Nivel 3: Definido. El proceso del software de las actividades de gestión y de ingeniería se documenta, se estandariza y se integra dentro de un proceso de software de toda una organización. Todos los proyectos utilizan una versión documentada y aprobada del proceso de la organización para el desarrollo y mantenimiento del software. En este nivel se incluyen todas las características definidas para el nivel 2.

Nivel 4: Gestionado. Se recopilan medidas detalladas del proceso del software y de la calidad del producto. Mediante la utilización de medidas detalladas, se comprenden y se controlan cuantitativamente tanto los productos como el proceso del software. En este nivel se incluyen todas las características definidas para el nivel 3.

Nivel 5: Optimización. Mediante una retroalimentación cuantitativa del proceso, ideas y tecnologías innovadoras se posibilita una mejora del proceso. En este nivel se incluyen todas las características definidas para el nivel 4.



El IIS ofrece un amplio conjunto de información relacionada con el proceso en **www.sei.cmu.edu**

Merece la pena destacar que cada nivel superior es la suma de los niveles anteriores, por ejemplo, un desarrollador de software en el nivel 3 tiene que haber alcanzado el estado nivel 2 para poder disponer de sus procesos en el nivel 3.

El nivel 1 representa una situación sin ningún esfuerzo en la garantía de calidad y gestión del proyecto, donde cada equipo del proyecto puede desarrollar software de cualquier forma eligiendo los métodos, estándares y procedimientos a utilizar que podrán variar desde lo mejor hasta lo peor.

El nivel 2 representa el hecho de que un desarrollador de software ha definido ciertas actividades tales como el informe del esfuerzo y del tiempo empleado, y el informe de las tareas realizadas.

El nivel 3 representa el hecho de que un desarrollador de software ha definido tanto procesos técnicos como de gestión, por ejemplo un estándar para la programación ha sido detallado y se hace cumplir por medio de procedimientos tales como auditorías. Este nivel es aquel en el que la mayoría de los desarrolladores de software, pretenden conseguir con estándares como el **ISO** 9001, y existen pocos casos de desarrolladores de software que superan este nivel.

El nivel 4 comprende el concepto de medición y el uso de métricas. El capítulo 4 describe este tema con más detalle. Sin embargo, cabe destacar el concepto de métrica para comprender la importancia que tiene que el desarrollador del software alcance el nivel 4 o el nivel 5.

Una métrica es una cantidad insignificante que puede extraerse de algún documento o código dentro de un proyecto de software. Un ejemplo de métrica es el número de ramas condicionales en una sección de código de un programa. Esta métrica es significativa en el sentido de que proporciona alguna indicación del esfuerzo necesario para probar el código: está directamente relacionado con el número de caminos de prueba dentro del código.

Una organización del nivel 4 maneja numerosas métricas. Estas métricas se utilizan entonces para supervisar y controlar un proyecto de software, por ejemplo:

- Una métrica de prueba puede usarse para determinar cuándo finalizar la prueba de un elemento del código.
- Una métrica de legilibilidad puede usarse para juzgar la legilibilidad de algún documento en lenguaje natural.
- Una métrica de comprensión del programa puede utilizarse para proporcionar algún umbral numérico que los programadores no pueden cruzar.

Para que estas métricas alcancen este nivel es necesario que todos los componentes del nivel 3 CMM, en castellano MCM (Modelo de Capacidad de Madurez), estén conseguidos, por ejemplo notaciones bien definidas para actividades como la especificación del diseño de requisitos, por lo que estas métricas pueden ser fácilmente extraídas de modo automático.

El nivel 5 es el nivel más alto a alcanzar. Hasta ahora, muy pocos desarrolladores de software han alcanzado esta fase. Representa la analogía del software con los mecanismos de control de calidad que existen en otras industrias de mayor madurez. Por ejemplo el fabricante de un producto industrial como un cojinete de bolas (rodamiento) puede supervisar y controlar la calidad de los rodamientos producidos y puede predecir esta calidad basándose en los procesos y máquinas utilizados para desarrollar los rodamientos. Del mismo modo que el desarrollador del sofware en el nivel 5 puede predecir resultados como el número de errores latentes en un producto basado en la medición tomada durante la ejecución de un proyecto. Además, dicho desarrollador puede cuantificar el efecto que un proceso nuevo o herramienta de manufacturación ha tenido en un proyecto examinando métricas para ese proyecto y comparándolas con proyectos anteriores que no utilizaron ese proceso o herramienta.

En este orden debe destacarse que para que un desarrollador de software alcance el nivel 5 tiene que tener cada proceso definido rigurosamente y seguirlo al pie de la letra; esto es una consecuencia de estar en el nivel 3. Si el desarrollador del software no tiene definidos rigurosamente los procesos pueden ocurrir una gran cantidad de cambios en el proceso de desarrollo y no se podrán utilizar las estadísticas para estas actividades.

Los cinco niveles definidos por el SEI se obtienen como consecuencia de evaluar las respuestas del cuestionario de evaluación basado en el MCM *(Modelo de capacidad de madurez)*. Los resultados del cuestionario se refinan en un único grado numérico que proporciona una indicación de la madurez del proceso de una organización.

El SEI ha asociado áreas claves del proceso (ACPs) a cada uno de los niveles de madurez. Las ACPs describen esas funciones de la ingeniería del software (por ejemplo: planificación del proyecto de software, gestión de requisitos) que se deben presentar para satisfacer una buena práctica a un nivel en particular. Cada ACP se describe identificando las características siguientes:



lodo organización debería esforzarse poro alcanzar lo profundidad del MCM del IIS. Sin embargo, lo implementación de cualquier aspecto del modelo puede eliminarse en su situación.

- *Objetivos* — los objetivos globales que debe alcanzar la ACP
- *Compromisos* — requisitos (impuestos en la organización) que se deben cumplir para lograr los objetivos y que proporcionan una prueba del intento por ajustarse a los objetivos.
- *Capacidades* — aquellos elementos que deben encontrarse (organizacional y técnicamente) para permitir que la organización cumpla los objetivos.
- *Actividades* — las tareas específicas que se requieren para lograr la función ACP.
- *Métodos para supervisar la implementación* — *la* manera en que las actividades son supervisadas conforme se aplican.
- *Métodos para verificar la implementución* — la forma en que se puede verificar la práctica adecuada para la ACP.

Se definen dieciocho ACPs (descritas mediante la estructura destacada anteriormente) en el modelo de madurez y se distribuyen en niveles diferentes de madurez del proceso. Las ACPs se deberían lograr en cada nivel de madurez de proceso[4]:



Una versión tabular del MCM completo del IIS, incluyendo todos los objetivos, comentarios, habilidades y actividades está disponible en
sepo.nosc.mil/CMMmatrices.html

Nivel 2 de Madurez del Proceso
- Gestión de configuración del software
- Garantía de calidad del software
- Gestión de subcontratación del software
- Seguimiento y supervisión del proyecto del software
- Planificación del proyecto del software
- Gestión de requisitos

Nivel 3 de Madurez del Proceso
- Revisiones periódicas
- Coordinación entre grupos
- Ingeniería de productos de software
- Gestión de integración del software
- Programa de formación
- Definición del proceso de la organización
- Enfoque del proceso de la organización

Nivel 4 de Madurez del Proceso
- Gestión de calidad del software
- Gestión cuantitativa del proceso

Nivel 5 *de Madurez del Proceso*
- Gestión de cambios del proceso
- Gestión de cambios de tecnología
- Prevención de defectos

Cada una de las ACPs se definen con un conjunto de *prácticas clave* que contribuyen a cumplir estos objetivos. Las prácticas clave son normas, procedimientos y actividades que deben ocurrir antes de que se haya instituido completamente un área clave de proceso. El SEI define a los *indicadores clave* como «aquellas prácticas clave o componentes de prácticas clave que ofrecen una visión mejor para lograr los objetivos de un área clave de proceso». Las cuestiones de valoración se diseñan para averiguar la existencia (o falta) de un indicador clave.

[4] Téngase en cuenta que las ACPs son acumulativas. Por ejemplo, el nivel *3* de madurez del proceso contiene todas las ACPs del nivel 2 más las destacadas para el nivel 1.

2.3 MODELOS DE PROCESO DEL SOFTWARE

Para resolver los problemas reales de una industria, un ingeniero del software o un equipo de ingenieros debe incorporar una estrategia de desarrollo que acompañe al proceso, métodos y capas de herramientas descritos en la Sección 2.1.1 y las fases genéricas discutidas en la Sección 2.1.2. Esta estrategia a menudo se llama *modelo de proceso o paradigma de ingeniería del software.* Se selecciona un modelo de proceso para la ingeniería del software según la naturaleza del proyecto y de la aplicación, los métodos y las herramientas a utilizarse, y los controles y entregas que se requieren. En un documento intrigante sobre la naturaleza del proceso del software, L.B.S. Raccoon [RAC95] utiliza fractales como base de estudio de la verdadera naturaleza del proceso del software.

Cita:

Frecuentemente, el trabajo del software sigue la primera ley del ciclismo: no importa dónde vayas, ya sea montaña arriba y contra el viento.

Anónimo

Todo el desarrollo del software se puede caracterizar como bucle de resolución de problemas (Fig. 2.3a) en el que se encuentran cuatro etapas distintas: «status quo», definición de problemas, desarrollo técnico e integración de soluciones. Status quo «representa el estado actual de sucesos» [RAC95]; la definición de problemas identifica el problema específico a resolverse; el desarrollo técnico resuelve el problema a través de la aplicación de alguna tecnología y la integración de soluciones ofrece los resultados (por ejemplo: documentos, programas, datos, nueva función comercial, nuevo producto) a los que solicitan la solución en primer lugar. Las fases y los pasos genéricos de ingeniería del software definidos en la Sección 2.1.2 se divide fácilmente en estas etapas.

En realidad, es difícil compartimentar actividades de manera tan nítida como la Figura 2.3.b da a entender, porque existen interferencias entre las etapas. Aunque esta visión simplificada lleva a una idea muy importante: con independencia del modelo de proceso que se seleccione para un proyecto de software, todas las etapas —status quo, definición de problemas, desarrollo técnico e integración de soluciones— coexisten simultáneamente en algún nivel de detalle. Dada la naturaleza recursiva de la Figura 2.3b, las cuatro etapas tratadas anteriormente se aplican igualmente al análisis de una aplicación completa y a la generación de un pequeño segmento de código.

Raccoon [RAC95] sugiere un «Modelo del Caos» que describe el «desarrollo del software como una extensión desde el usuario hasta el desarrollador y la tecnología». Conforme progresa el trabajo hacia un sistema completo, las etapas descritas anteriormente se aplican recursivamente a las necesidades del usuario y a la especificación técnica del desarrollador del software.



Todas las etapas de un proceso del software - estado actual, definición del problema, desarrollo técnico e integración de la solución — coexisten simultáneamente en algún nivel de detalle.

En las secciones siguientes, se tratan diferentes modelos de procesos para la ingeniería del software. Cada una representa un intento de ordenar una actividad inherentemente caótica. Es importante recordar que cada uno de los modelos se han caracterizado de forma que ayuden (con esperanza) al control y a la coordinación de un proyecto de software real. Y a pesar de eso, en el fondo, todos los modelos exhiben características del «Modelo del Caos».

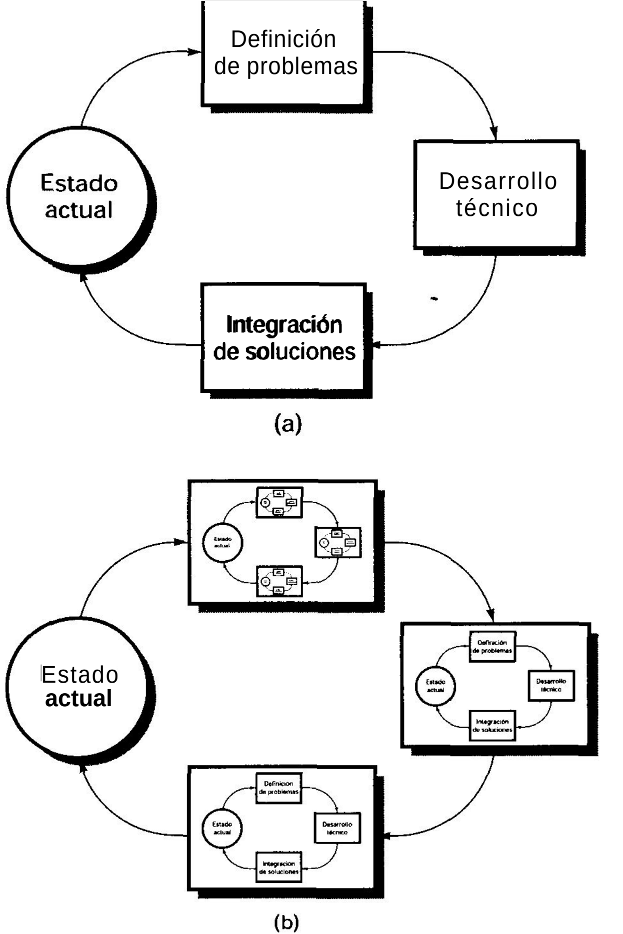


FIGURA 2.3. a) Las fases de un bucle de resolución de problemas [RAC 951. b) Fases dentro de las fases del bucle de resolución de problemas [RAC 95].

2.4 EL MODELO LINEAL SECUENCIAL

Llamado algunas veces «ciclo de vida básico» o «modelo en cascada», el *modelo lineal secuencial* sugiere un enfoque[5] sistemático, secuencial, para el desarrollo del software que comienza en un nivel de sistemas y progresa con el análisis, diseño, codificación, pruebas y mantenimiento. La Figura 2.4 muestra el modelo lineal secuencial para la ingeniería del software. Modelado según el ciclo de ingeniería convencional, el modelo lineal secuencial comprende las siguientes actividades:

Ingeniería y modelado de Sistemas/Información. Como el software siempre forma parte de un sistema más grande (o empresa), el trabajo comienza estableciendo requisitos de todos los elementos del sistema y asignando al software algún subgrupo de estos requisitos. Esta visión del sistema es esencial cuando el software se debe interconectar con otros elementos como hardware, personas y bases de datos. La ingeniería y el análisis de sistemas comprende los requisitos que se recogen en el nivel del sistema con una pequeña parte de análisis y de diseño. La ingeniería de información abarca los requisitos que se recogen en el nivel de empresa estratégico y en el nivel del área de negocio.

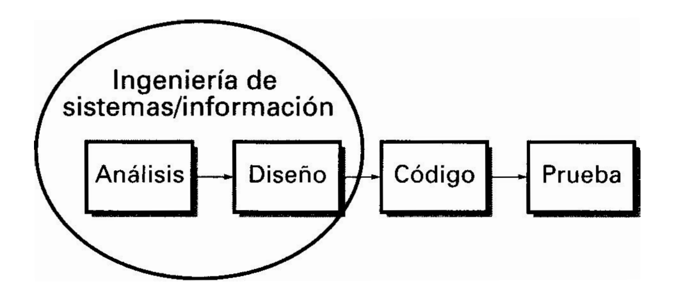


FIGURA 2.4. El modelo lineal secuencial.

Análisis de los requisitos del software. El proceso de reunión de requisitos se intensifica y se centra especialmente en el software. Para comprender la naturaleza del (los) programa(s) a construirse, el ingeniero («analista») del software debe comprender el dominio de información del software (descrito en el Capítulo 11), así como la función requerida, comportamiento, rendimiento e interconexión.

Diseño. El diseño del software es realmente un proceso de muchos pasos que se centra en cuatro atributos distintos de programa: estructura de datos, arquitectura de software, representaciones de interfaz y detalle procedimental (algoritmo). El proceso del diseño traduce requisitos en una representación del software donde se pueda evaluar su calidad antes de que comience la codificación.

Generación de código. El diseño se debe traducir en una forma legible por la máquina. El paso de generación de código lleva a cabo esta tarea. Si se lleva a cabo el diseño de una forma detallada, la generación de código se realiza mecánicamente.

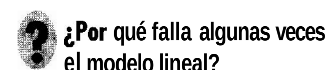


Aunque el modelo lineal es a menudo denominado «modelo tradicional», resulta un enfoque razonable cuando los requisitos se han entendido correctamente.

Pruebas. Una vez que se ha generado el código, comienzan las pruebas del programa. El proceso de pruebas se centra en los procesos lógicos internos del software, asegurando que todas las sentencias se han comprobado, y en los procesos externos funcionales; es decir, realizar las pruebas para la detección de errores y asegurar que la entrada definida produce resultados reales de acuerdo con los resultados requeridos.

Mantenimiento. El software indudablemente sufrirá cambios después de ser entregado al cliente (una excepción posible es el software empotrado). Se producirán cambios porque se han encontrado errores, porque el software debe adaptarse para acoplarse a los cambios de su entorno externo (por ejemplo: se requiere un cambio debido a un sistema operativo o dispositivo periférico nuevo), o porque el cliente requiere mejoras funcionales o de rendimiento. El soporte y mantenimiento del software vuelve a aplicar cada una de las fases precedentes a un programa ya existente y no a uno nuevo.

El modelo lineal secuencial es el paradigma más antiguo y más extensamente utilizado en la ingeniería del software. Sin embargo, la crítica del paradigma ha puesto en duda su eficacia [HAN95]. Entre los problemas que se encuentran algunas veces en el modelo lineal secuencial se incluyen:



1. Los proyectos reales raras veces siguen el modelo secuencial que propone el modelo. Aunque el modelo lineal puede acoplar interacción, lo hace indirectamente. Como resultado, los cambios pueden causar confusión cuando el equipo del proyecto comienza.

[5] Aunque el modelo original en cascada propuesto por Winston Royce [ROY70] hacía provisiones para «bucles de realimentación», la gran mayoría de las organizaciones que aplican este modelo de proceso lo hacen como si fuera estrictamente lineal.

2. A menudo es difícil que el cliente exponga explícitamente todos los requisitos. El modelo lineal secuencial lo requiere y tiene dificultades a la hora de acomodar la incertidumbre natural al comienzo de muchos proyectos.
3. El cliente debe tener paciencia. Una versión de trabajo del (los) programa(s) no estará disponible hasta que el proyecto esté muy avanzado. Un grave error puede ser desastroso si no se detecta hasta que se revisa el programa.

Cada uno de estos errores es real. Sin embargo, el paradigma del ciclo de vida clásico tiene un lugar definido e importante en el trabajo de la ingeniería del software. Proporciona una plantilla en la que se encuentran métodos para análisis, diseño, codificación, pruebas y mantenimiento. El ciclo de vida clásico sigue siendo el modelo de proceso más extensamente utilizado por la ingeniería del software. Pese a tener debilidades, es significativamente mejor que un enfoque hecho al *azar* para el desarrollo del software.

2.5 EL MODELO DE CONSTRUCCIÓN DE PROTOTIPOS

Un cliente, a menudo, define un conjunto de objetivos generales para el software, pero no identifica los requisitos detallados de entrada, proceso o salida. En otros casos, el responsable del desarrollo del software puede no estar seguro de la eficacia de un algoritmo, de la capacidad de adaptación de un sistema operativo, o de la forma en que debería tomarse la interacción hombre-máquina. En estas y en otras muchas situaciones, un *paradigma de construcción **de** prototipos* puede ofrecer el mejor enfoque.

El paradigma de construcción de prototipos (Fig. 2.5) comienza con la recolección de requisitos. El desarrollador y el cliente encuentran y definen los objetivos globales para el software, identifican los requisitos conocidos y las áreas del esquema en donde es obligatoria más definición. Entonces aparece un «diseño rápido». El diseño rápido se centra en una representación de esos aspectos del software que serán visibles para el usuario/cliente (por ejemplo: enfoques de entrada y formatos de salida). El diseño rápido lleva a la construcción de un prototipo. El prototipo lo evalúa el cliente/usuario y se utiliza para refinar los requisitos del software a desarrollar. La iteración ocurre cuando el prototipo se pone a punto para satisfacer las necesidades del cliente, permitiendo al mismo tiempo que el desarrollador comprenda mejor lo que se necesita hacer.

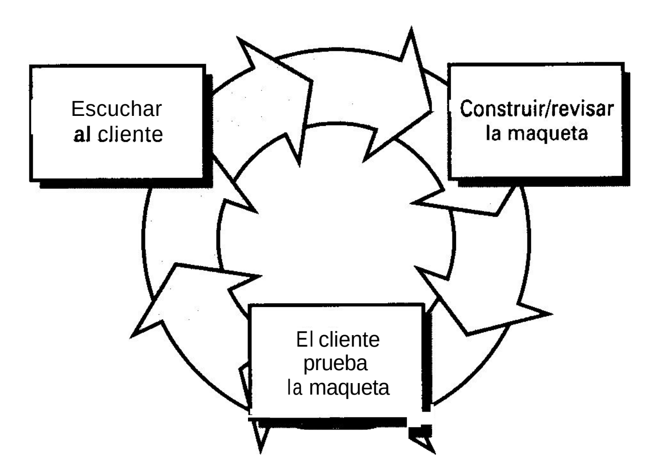


FIGURA 2.5. El paradigma de construcción de prototipos.



Cuando el cliente tiene una necesidad legítima, pero está desorientado sobre los detalles, el primer paso es desarrollar un prototipo.

Lo ideal sería que el prototipo sirviera como un mecanismo para identificar los requisitos del software. Si se construye un prototipo de trabajo, el desarrollador intenta hacer uso de los fragmentos del programa ya existentes o aplica herramientas (por ejemplo: generadores de informes, gestores de ventanas, etc.) que permiten generar rápidamente programas de trabajo.

Pero ¿qué hacemos con el prototipo una vez que ha servido para el propósito descrito anteriormente? Brooks [BRO75] proporciona una respuesta:

> **En la mayoría de los proyectos, el primer sistema construido apenas se puede utilizar. Puede ser demasiado lento, demasiado grande o torpe en su uso, o las tres a la vez. No hay otra alternativa que comenzar de nuevo, aunque nos duela pero es más inteligente, y construir una versión rediseñada en la que se resuelvan estos problemas ... Cuando se utiliza un concepto nuevo de sistema o una tecnología nueva, se tiene que construir un sistema que no sirva y se tenga que tirar, porque incluso la mejor planificación no es omnisciente como para que esté perfecta la primera vez. Por lo tanto la pregunta de la gestión no es si construir un sistema piloto y tirarlo. Tendremos que hacerlo. La Única pregunta es si planificar de antemano construir un desechable, o prometer entregárselo a los clientes...**

El prototipo puede servir como «primer sistema». El que Brooks recomienda tirar. Aunque esta puede ser una visión idealizada. Es verdad que a los clientes y a los que desarrollan les gusta el paradigma de construcción de prototipos. A los usuarios les gusta el sistema real y a los que desarrollan les gusta construir algo inmediatamente. Sin embargo, la construcción de prototipos también puede ser problemática por las siguientes razones:

1. El cliente ve lo que parece ser una versión de trabajo del software, sin tener conocimiento de que el prototipo también está junto con «el chicle y el cable de embalar», sin saber que con la prisa de hacer que funcione no se ha tenido en cuenta la calidad del software global o la facilidad de mantenimiento a largo plazo. Cuando se informa de que el producto se debe construir otra vez para que se puedan mantener los niveles altos de calidad, el cliente no lo entiende y pide que se apliquen «unos pequeños ajustes» para que se pueda hacer del prototipo un producto final. De forma demasiado frecuente la gestión de desarrollo del software es muy lenta.
2. El desarrollador, a menudo, hace compromisos de implementación para hacer que el prototipo funcione rápidamente. Se puede utilizar un sistema operativo o lenguaje de programación inadecuado simplemente porque está disponible y porque es conocido; un algoritmo eficiente se puede implementar simplemente para demostrar la capacidad. Después de algún tiempo, el desarrollador debe familiarizarse con estas selecciones, y olvidarse de las razones por las que son inadecuadas. La selección menos ideal ahora es una parte integral del sistema.



Resisto lo presión de ofrecer un mal prototipo en el producto final. Como resultado, lo colidod se resiente casi siempre.

Aunque pueden surgir problemas, la construcción de prototipos puede ser un paradigma efectivo para la ingeniería del software. La clave es definir las reglas del juego al comienzo; es decir, el cliente y el desarrollador se deben poner de acuerdo en que el prototipo se construya para servir como un mecanismo de definición de requisitos.

2.6 EL MODELO DRA*

El *Desarrollo Rápido de Aplicaciones* (**DRA**) es un modelo de proceso del desarrollo del software lineal secuencial que enfatiza un ciclo de desarrollo extremadamente corto. El modelo DRA es una adaptación a «alta velocidad» del modelo lineal secuencial en el que se logra el desarrollo rápido utilizando una construcción basada en componentes. Si se comprenden bien los requisitos y se limita el ámbito del proyecto, el proceso DRA permite al equipo de desarrollo crear un «sistema completamente funcional» dentro de períodos cortos de tiempo (por ejemplo: de 60 a 90 días) [MAR91]. Cuando se utiliza principalmente para aplicaciones de sistemas de información, el enfoque DRA comprende las siguientes fases [KER94]:

Modelado de Gestión. El flujo de información entre las funciones de gestión se modela de forma que responda a las siguientes preguntas: ¿Qué información conduce el proceso de gestión? ¿Qué información se genera? ¿Quién la genera? ¿A dónde va la información? ¿Quién la procesa? El modelado de gestión se describe con más detalle en el Capítulo 10.

Modelado de datos. El flujo de información definido como parte de la fase de modelado de gestión se refina como un conjunto de objetos de datos necesarios para apoyar la empresa. Se definen las características (llamadas atributos) de cada uno de los objetos y las relaciones entre estos objetos. El modelado de datos se trata en el Capítulo 12.

Modelado del proceso. Los objetos de datos definidos en la fase de modelado de datos quedan transformados para lograr el flujo de información necesario para implementar una función de gestión. Las descripciones del proceso se crean para añadir, modificar, suprimir, o recuperar un objeto de datos.

Generación de aplicaciones. El DRA asume la utilización de técnicas de cuarta generación (Sección 2.10). En lugar de crear software con lenguajes de programación de tercera generación, el proceso DRA trabaja para volver a utilizar componentes de programas ya existentes (cuando es posible) o a crear componentes reutilizables (cuando sea necesario). En todos los casos se utilizan herramientas para facilitar la construcción del software.

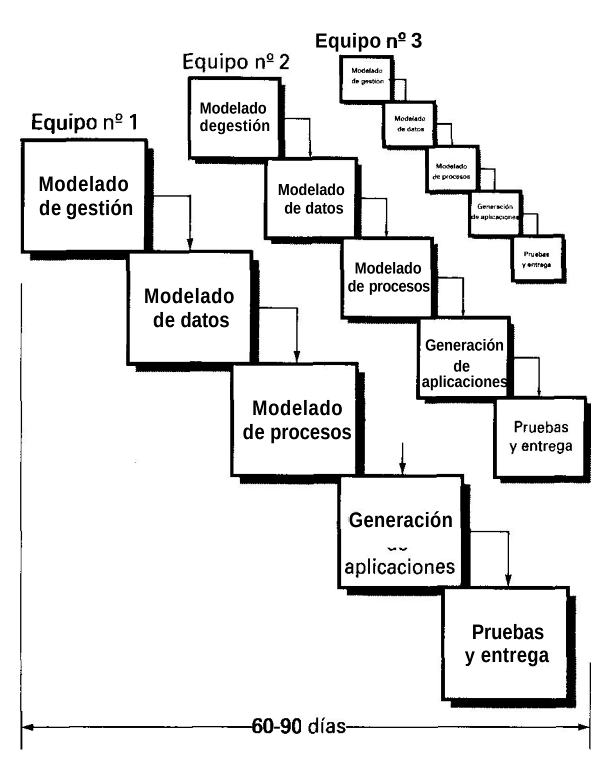


FIGURA 2.6. El modelo **DRA.**

* En ingles: RAD (Rapid Application Deveiopment)

Pruebas y entrega. Como el proceso **DRA** enfatiza la reutilización, ya se han comprobado muchos de los componentes de los programas. Esto reduce tiempo de pruebas. Sin embargo, se deben probar todos los componentes nuevos y se deben ejercitar todas las interfaces a fondo.

Referencia cruzada

El DRA hace un fuerte uso de componentes reutilizables. Para mayor información sobre el desarrollo de componentes, véase el Capítulo 27.

El modelo de proceso **DRA** se ilustra en la Figura 2.6. Obviamente, las limitaciones de tiempo impuestas en un proyecto **DRA** demandan «ámbito en escalas» [KER94]. Si una aplicación de gestión puede modularse de forma que permita completarse cada una de las funciones principales en menos de tres meses (utilizando el enfoque descrito anteriormente), es un candidato del DRA. Cada una de las funciones pueden ser afrontadas por un equipo **DRA** separado y ser integradas en un solo conjunto.

Al igual que todos los modelos de proceso, el enfoque **DRA** tiene inconvenientes [BUT94]:

- Para proyectos grandes aunque por escalas, el **DRA** requiere recursos humanos suficientes como para crear el número correcto de equipos **DRA.**
- **DRA** requiere clientes y desarrolladores comprometidos en las rápidas actividades necesarias para completar un sistema en un marco de tiempo abreviado. Si no hay compromiso por ninguna de las partes constituyentes, los proyectos **DRA** fracasarán.
- No todos los tipos de aplicaciones son apropiados para **DRA.** Si un sistema no se puede modularizar adecuadamente, la construcción de los componentes necesarios para **DRA** será problemático. **Si** está en juego el alto rendimiento, y se va a conseguir el rendimiento convirtiendo interfaces en componentes de sistemas, el enfoque **DRA** puede que no funcione.
- **DRA** no es adecuado cuando los riesgos técnicos son altos. Esto ocurre cuando una nueva aplicación hace uso de tecnologías nuevas, o cuando el software nuevo requiere un alto grado de interoperatividad con programas de computadora ya existentes.

2.7 MODELOS EVOLUTIVOS DE PROCESO DEL SOFTWARE

Se reconoce que el software, al igual que todos los sistemas complejos, evoluciona con el tiempo [GIL88]. Los requisitos de gestión y de productos a menudo cambian conforme a que el desarrollo proceda haciendo que el camino que lleva al producto final no sea real; las estrictas fechas tope del mercado hacen que sea imposible finalizar un producto completo, por lo que se debe introducir una versión limitada para cumplir la presión competitiva **y** de gestión; se comprende perfectamente el conjunto de requisitos de productos centrales o del sistema, pero todavía se tienen que definir los detalles de extensiones del producto o sistema. En estas y en otras situaciones similares, los ingenieros del software necesitan un modelo de proceso que se ha diseñado explícitamente para acomodarse a un producto que evolucione con el tiempo.

El modelo lineal secuencial (Sección 2.4) se diseña para el desarrollo en línea recta. En esencia, este enfoque en cascada asume que se va entregar un sistema completo una vez que la secuencia lineal se haya finalizado. El modelo de construcción de prototipos (Sección 2.5) se diseña para ayudar al cliente (o al que desarrolla) a comprender los requisitos. En general, no se diseña para entregar un sistema de producción. En ninguno de los paradigmas de ingeniería del software se tiene en cuenta la naturaleza evolutiva del software.

Los modelos evolutivos son iterativos. Se caracterizan por la forma en que permiten a los ingenieros del software desarrollar versiones cada vez más completas del software.

2.7.1. El modelo incremental

El *modelo incrernental* combina elementos del modelo lineal secuencial (aplicados repetidamente) con la filosofía interactiva de construcción de prototipos. Como muestra la Figura 2.7, el modelo incremental aplica secuencias lineales de forma escalonada mientras progresa el tiempo en el calendario. Cada secuencia lineal produce un «incremento» del software [MDE93]. Por ejemplo, el software de tratamiento de textos desarrollado con el paradigma incremental podría extraer funciones de gestión de archivos básicos y de producción de documentos en el primer incremento; funciones de edición más sofisticadas y de producción de documentos en el segundo incremento; corrección ortográfica **y** gramatical en el tercero; y una función avanzada de esquema de página en el cuarto. Se debería tener en cuenta que el flujo del proceso de cualquier incremento puede incorporar el paradigma de construcción de prototipos.



El modelo incremental entrega el software en partes pequeñas, pero utilizables, llamadas ((incrementos). En general, cado incremento se construye sobre aquél que ya ho sido entregado.

Cuando se utiliza un modelo incremental, el primer incremento a menudo es un producto esencial. **Es** decir, se afrontan requisitos básicos, pero muchas funciones

suplementarias (algunas conocidas, otras no) quedan sin extraer. El cliente utiliza el producto central (o sufre la revisión detallada). Como un resultado de utilización y/o de evaluación, se desarrolla un plan para el incremento siguiente. El plan afronta la modificación del producto central a fin de cumplir mejor las necesidades del cliente y la entrega de funciones, y características adicionales. Este proceso se repite siguiendo la entrega de cada incremento, hasta que se elabore el producto completo.



Cuando tenga una fecha de entrega imposible de cambiar, el modela incremental es un buen paradigma a considerar.

El modelo de proceso incremental, como la construcción de prototipos (Sección *2.5)* **y** otros enfoques evolutivos, es iterativo por naturaleza. Pero a diferencia de la construcción de prototipos, el modelo incremental se centra en la entrega de un producto operacional con cada incremento. Los primeros incrementos son versiones «incompletas» del producto final, pero proporcionan al usuario la funcionalidad que precisa y también una plataforma para la evaluación.

El desarrollo incremental es particularmente útil cuando la dotación de personal no está disponible para una implementación completa en la fecha límite que se haya establecido para el proyecto. Los primeros incrementos se pueden implementar con menos personas.

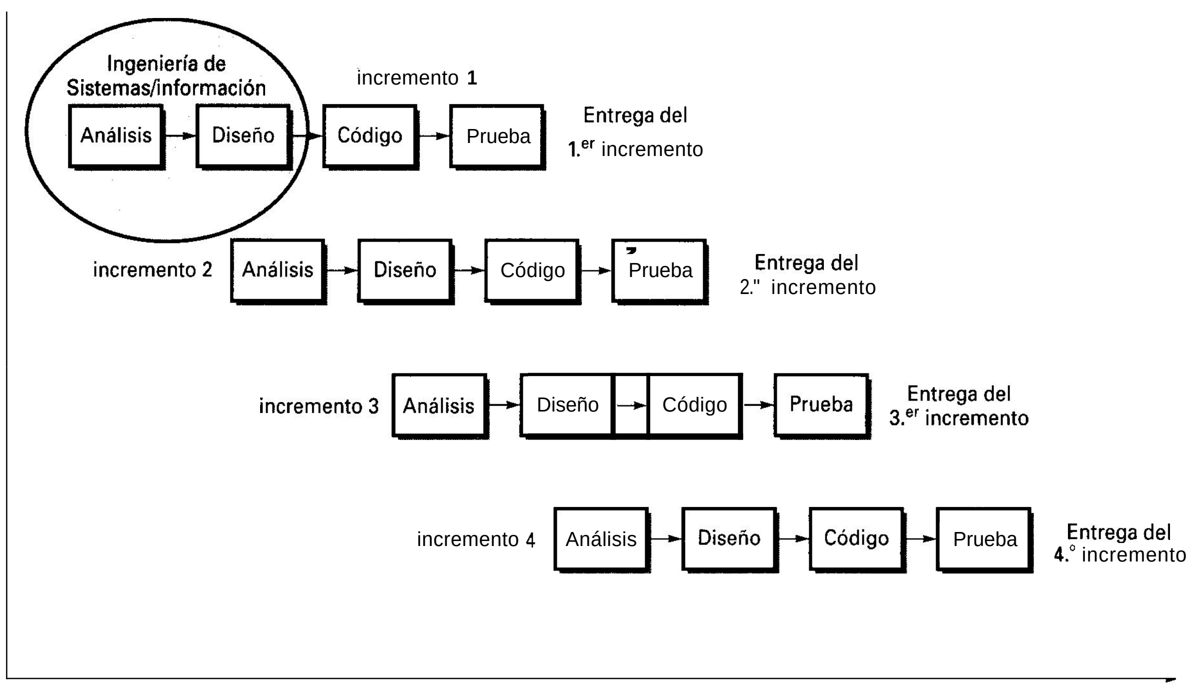


FIGURA 2.7. El modelo incremental.

2.7.2. El modelo espiral

El *modelo en espiral*, propuesto originalmente por Boehm [BOE88], es un modelo de proceso de software evolutivo que conjuga la naturaleza iterativa de construcción de prototipos con los aspectos controlados y sistemáticos del modelo lineal secuencial. Proporciona el potencial para el desarrollo rápido de versiones incrementales del software. En el modelo espiral, el software se desarrolla en una serie de versiones incrementales. Durante las primeras iteracciones, la version incremental podría ser un modelo en papel o un prototipo. Durante las últimas iteraciones, se producen versiones cada vez más completas del sistema diseñado.

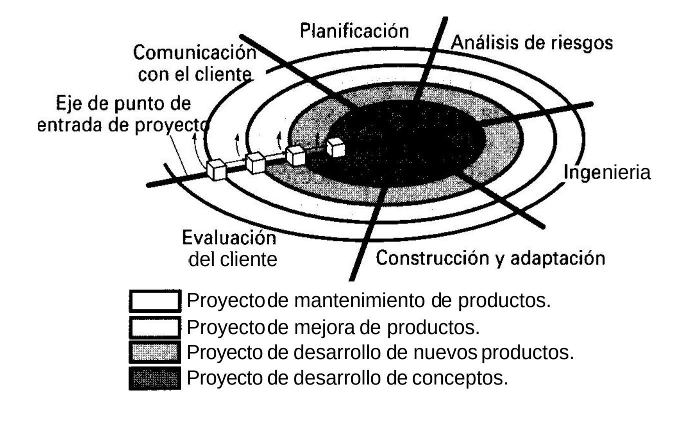


FIGURA 2.8. Un modelo espiral típico.

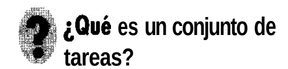
El modelo en espiral se divide en un número de actividades de marco de trabajo, también llamadas *regiones de tareas*[6]. Generalmente, existen entre tres y seis regiones de tareas. La Figura 2.8 representa un modelo en espiral que contiene seis regiones de tareas:

- **comunicación con el cliente —** las tareas requeridas para establecer comunicación entre el desarrollador y el cliente.
- **planificación —** las tareas requeridas para definir recursos, el tiempo y otra información relacionadas con el proyecto.
- **análisis de riesgos —** las tareas requeridas para evaluar riesgos técnicos y de gestión.
- **ingeniería —** las tareas requeridas para construir una o más representaciones de la aplicación.
- **construcción y acción —** las tareas requeridas para construir, probar, instalar y proporcionar soporte al usuario (por ejemplo: documentación y práctica)
- **evaluación del cliente —** las tareas requeridas para obtener la reacción del cliente según la evaluación de las representaciones del software creadas durante la etapa de ingeniería e implementada durante la etapa de instalación.

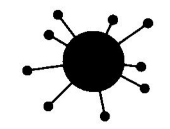


Los actividades del marco de trabajo se aplican a cualquier proyecto de software que realice, sin tener en cuenta el tamaño ni la complejidad.

Cada una de las regiones están compuestas por un conjunto de tareas del trabajo, llamado *conjunto de tareas*, que se adaptan a las características del proyecto que va a emprenderse. Para proyectos pequeños, el número de tareas de trabajo y su formalidad es bajo. Para proyectos mayores y más críticos cada región de tareas contiene tareas de trabajo que se definen para lograr un nivel más alto de formalidad. En todos los casos, se aplican las actividades de protección (por ejemplo: gestión de configuración del software y garantía de calidad del software) mencionadas en la Sección 2.2.



Cuando empieza este proceso evolutivo, el equipo de ingeniería del software gira alrededor de la espiral en la dirección de las agujas del reloj, comenzando por el centro. El primer circuito de la espiral puede producir el desarrollo de una especificación de productos; los pasos siguientes en la espiral se podrían utilizar para desarrollar un prototipo y progresivamente versiones más sofisticadas del software. Cada paso por la región de planificación produce ajustes en el plan del proyecto. El coste y la planificación se ajustan con la realimentacion ante la evaluación del cliente. Además, el gestor del proyecto ajusta el número planificado de iteraciones requeridas para completar el software.



Modelo de proceso adaptable.

A diferencia del modelo de proceso clásico que termina cuando se entrega el software, el modelo en espiral puede adaptarse y aplicarse a lo largo de la vida del software de computadora. Una visión alternativa del modelo en espiral puede ser considerada examinando el *eje de punto de entrada en el proyecto* también reflejado en la Figura 2.8. Cada uno de los cubos situados a lo largo del eje pueden usarse para representar el punto de arranque para diferentes tipos de proyectos. Un «proyecto de desarrollo de conceptos» comienza en el centro de la espiral y continuará (aparecen múltiples iteraciones a lo largo de la espiral que limita la región sombreada central) hasta que se completa el desarrollo del concepto. Si el concepto se va a desarrollar dentro de un producto real, el proceso continúa a través del cubo siguiente (punto de entrada del proyecto de desarrollo del producto nuevo) y se inicia un «nuevo proyecto de desarrollo". El producto nuevo evolucionará a través de iteraciones alrededor de la espiral siguiendo el camino que limita la región algo más brillante que el centro. En esencia, la espiral, cuando se caracteriza de esta forma, permanece operativa hasta que el software se retira. Hay veces en que el proceso está inactivo, pero siempre que se inicie un cambio, el proceso arranca en el punto de entrada adecuado (por ejemplo: mejora del producto).

El modelo en espiral es un enfoque realista del desarrollo de sistemas y de software a gran escala. Como el software evoluciona, a medida que progresa el proceso el desarrollador y el cliente comprenden y reaccionan mejor ante riesgos en cada uno de los niveles evolutivos. El modelo en espiral utiliza la construcción de prototipos como mecanismo de reducción de riesgos, pero, lo que es más importante, permite a quien lo desarrolla aplicar el enfoque de construcción de prototipos en cualquier etapa de evolución del producto. Mantiene el enfoque sistemático de los pasos sugeridos por el ciclo de vida clásico, pero lo incorpora al marco de trabajo iterativo que refleja de forma más realista el mundo real. El modelo en espiral demanda una consideración direc-

[6] El modelo en espiral de esta sección es una variante sobre el modelo propuesto por Boehm. Para más información sobre el modelo en espiral original, consulte [BOE88]. Un tratado más reciente del modelo en espiral puede encontrarse en [BOE98].

ta de los riesgos técnicos en todas las etapas del proyecto, y, si se aplica adecuadamente, debe reducir los riesgos antes de que se conviertan en problemáticos.

Referencia cruzada

Modelos evolutivos como el modelo en espiral, son apropiados, particularmente, para el desarrollo de sistemas orientados a objetos. Para más detalles, véase la Parte cuarta.

Pero al igual que otros paradigmas, el modelo en espiral no es la panacea. Puede resultar difícil convencer a grandes clientes (particularmente en situaciones bajo contrato) de que el enfoque evolutivo es controlable. Requiere una considerable habilidad para la evaluación del riesgo, y cuenta con esta habilidad para el éxito. Si un riesgo importante no es descubierto y gestionado, indudablemente surgirán problemas. Finalmente, el modelo no se ha utilizado tanto como los paradigmas lineales secuenciales o de construcción de prototipos. Todavía tendrán que pasar muchos años antes de que se determine con absoluta certeza la eficacia de este nuevo e importante paradigma.

2.7.3. El modelo espiral WINWIN (Victoria&Victoria)

El modelo en espiral tratado en la Seccion 2.7.2 sugiere una actividad del marco de trabajo que aborda la comunicación con el cliente. El objetivo de esta actividad es mostrar los requisitos del cliente. En un contexto ideal, el desarrollador simplemente pregunta al cliente lo que se necesita y el cliente proporciona detalles suficientes para continuar. Desgraciadamente, esto raramente ocurre. En realidad el cliente y el desarrollador entran en un proceso de negociación, donde el cliente puede ser preguntado para sopesar la funcionalidad, rendimiento, y otros productos o características del sistema frente al coste y al tiempo de comercialización.

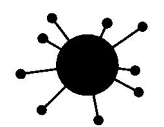


La obtención de requisitos del software requiere una negociación. Tiene éxito cuando ambas partes «ganan».

Las mejores negociaciones se esfuerzan en obtener[7] «victoria-victoria». Esto es, el cliente gana obteniendo el producto o sistema que satisface la mayor parte de sus necesidades y el desarrollador gana trabajando para conseguir presupuestos y lograr una fecha de entrega realista.

El modelo en espiral WINWIN de Boehm [BOE98] define un conjunto de actividades de negociación al principio de cada paso alrededor de la espiral. Más que una simple actividad de comunicación con el cliente, se definen las siguientes actividades:

1. Identificación del sistema o subsistemas clave de los «directivos»[8].
2. Determinación de las «condiciones de victoria» de los directivos.
3. Negociación de las condiciones de «victoria» de los directivos para reunirlas en un conjunto de condiciones «victoria-victoria» para todos los afectados (incluyendo el equipo del proyecto de software).



Habilidades de negociación

Conseguidos completamente estos pasos iniciales se logra un resultado «victoria-victoria», que será el criterio clave para continuar con la definición del sistema y del software. El modelo en espiral WINWIN se ilustra en la Figura 2.9.

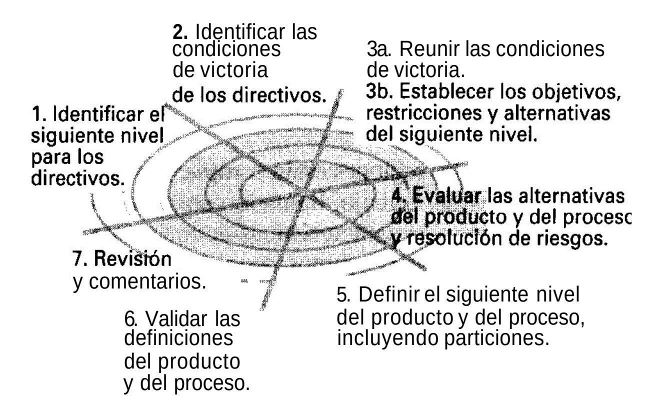


FIGURA 2.9. El modelo en espiral **WINWIN** [BOE98].

Además del énfasis realizado en la negociación inicial, el modelo en espiral WINWIN introduce tres hitos en el proceso, llamados ***puntos de** fijación* [BOE96], que ayudan a establecer la completitud de un ciclo alrededor de la espiral y proporcionan hitos de decisión antes de continuar el proyecto de software.

En esencia, los puntos de fijación representan tres visiones diferentes del progreso mientras que el pro-

[7] Hay docenas de libros escritos sobre habilidades en la negociación [p. ej., [FIS91], [DON96], [FAR97]].Es una de las cosas mas importantes que un ingeniero o gestor joven (oviejo) puede aprender. Lea uno.

[8] Un directivo es alguien en la organización que tiene un interés directo, por el negocio, en el sistema o producto a construir y puede ser premiado por un resultado con éxito o criticado si el esfuerzo falla.

yecto recorre la espiral. El primer punto de fijación, llamado *objetivos del ciclo de vida* (OCV), define un conjunto de objetivos para cada actividad principal de ingeniería del software. Como ejemplo, de una parte de OCV, un conjunto de objetivos asociados a la definición de los requisitos del producto/sistema del nivel más alto. El segundo punto de fijación, llamado *arquitectura del ciclo de vida* (ACV), establece los objetivos que se deben conocer mientras que se define la arquitectura del software y el sistema. Como ejemplo, de una parte de la ACV, el equipo del proyecto de software debe demostrar que ha evaluado la funcionalidad de los componentes del software reutilizables y que ha considerado su impacto en las decisiones de arquitectura. *La capacidad operativa inicial* (COI) es el tercer punto de fijación y representa un conjunto de objetivos asociados a la preparación del software para la instalación/distribución, preparación del lugar previamente a la instalación, y la asistencia precisada de todas las partes que utilizará o mantendrá el software.

2.7.4. El modelo de desarrollo concurrente

Davis y Sitaram [DAV94] han descrito el modelo de desarrollo concurrente, llamado algunas veces ingeniería concurrente, de la forma siguiente:

> Los gestores de proyectos que siguen los pasos del estado del proyecto en lo que se refiere a las fases importantes [del ciclo de vida clásico] no tienen idea del estado de **sus** proyectos. Estos son ejemplos de un intento por seguir **los** pasos extremadamente complejos de actividades mediante modelos demasiado simples. Tenga en cuenta que aunque un proyecto [grande] esté en la fase de codificación, hay personal de ese proyecto implicado en actividades asociadas generalmente a muchas fases de desarrollo simultáneamente. Por ejemplo, ... El personal está escribiendo requisitos, diseñando, codificando, haciendo pruebas y probando la integración [todo al mismo tiempo]. Los modelos de procesos de ingeniería del software de Humphrey y Kellner [HUM89, KEL89] han mostrado la concurrencia que existe para actividades que ocurren durante cualquier fase. El trabajo más reciente de Kellner [KEL91] utiliza diagramas de estado [una notación que representa los estados de un proceso] para representar la relación concurrente que existe entre actividades asociadas a un acontecimiento específico (por ejemplo: un cambio de requisitos durante el último desarrollo), pero falla en capturar la riqueza de la concurrencia que existe en todas las actividades de desarrollo y de gestión del software en un proyecto... La mayoría de los modelos de procesos de desarrollo del software son dirigidos por el tiempo; cuanto más tarde sea, más atrás se encontrará en el proceso de desarrollo. [Un modelo de proceso concurrente] está dirigido por las necesidades del usuario, las decisiones de la gestión y los resultados de las revisiones.

El modelo de proceso concurrente se puede representar en forma de esquema como una serie de actividades técnicas importantes, tareas y estados asociados a ellas. Por ejemplo, la actividad de ingeniería definida para el modelo en espiral (Sección 2.7.2), se lleva a cabo invocando las tareas siguientes: modelado de construcción de prototipos y/o análisis, especificación de requisitos y diseño'.

La Figura 2.10 proporciona una representación esquemática de una actividad dentro del modelo de proceso concurrente. La actividad —análisis— se puede encontrar en uno de los estados'" destacados anteriormente en cualquier momento dado. De forma similar, otras actividades (por ejemplo: diseño o comunicación con el cliente) se puede representar de una forma análoga. Todas las actividades existen concurrentemente, pero residen en estados diferentes. Por ejemplo, al principio del proyecto la actividad de *comunicación con el cliente* (no mostrada en la figura) ha finalizado su primera iteración y está en el estado de **cambios,en espera.** La actividad de *análisis* (que estaba en el estado **ninguno** mientras que se iniciaba la comunicación inicial con el cliente) ahora hace una transición al estado **bajo desarrollo.** Sin embargo, si el cliente indica que se deben hacer cambios en requisitos, la actividad *análisis* cambia del estado **bajo desarrollo** al estado **cambios en espera.**

El modelo de proceso concurrente define una serie de acontecimientos que dispararán transiciones de estado a estado para cada una de las actividades de la ingeniería del software. Por ejemplo, durante las primeras etapas del diseño, no se contempla una inconsistencia del modelo de análisis. Esto genera la *corrección del modelo de análisis* de sucesos, que disparará la actividad de *análisis* del estado **hecho** al estado **cambios en espera.**

El modelo de proceso concurrente se utiliza a menudo como el paradigma de desarrollo de aplicaciones cliente/servidor[11] (Capítulo 28). Un sistema cliente/servidor se compone de un conjunto de componentes funcionales. Cuando se aplica a cliente/servidor, el modelo de proceso concurrente define actividades en dos dimensiones [SHE94]: una dimensión de sistemas y una dimensión de componentes. Los aspectos del nivel de sistemas se afrontan mediante tres actividades: diseño, ensamblaje y uso.

[9] Se debería destacar que el análisis y el diseño son tareas complejas que requieren un estudio sustancial. Las Partes III y IV del libro consideran estos temas con más detalle.

[10] Un estado es algún modo externamente observable del comportamiento.

[11] En aplicaciones cliente/servidor, la funcionalidad del software se divide entre clientes (normalmente PCs) y un servidor (una computadora más potente) que generalmente mantiene una base de datos centralizada.

La dimensión de componentes se afronta con dos actividades: diseño y realización. La concurrencia se logra de dos formas: (1) las actividades de sistemas y de componentes ocurren simultáneamente y pueden modelarse con el enfoque orientado a objetos descrito anteriormente; (2) una aplicación cliente/servidor típica se implementa con muchos componentes, cada uno de los cuales se pueden diseñar y realizar concurrentemente. En realidad, el modelo de proceso concurrente es aplicable a todo tipo de desarrollo de software y proporciona una imagen exacta del estado actual de un proyecto.

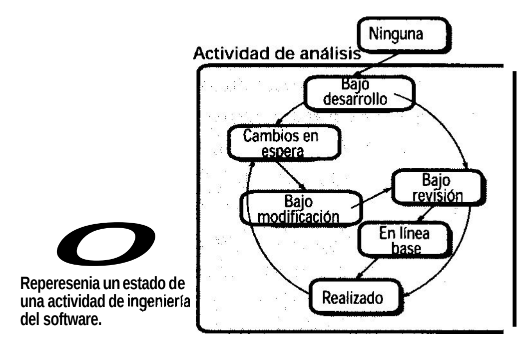


FIGURA 2.10. Un elemento del modelo de proceso concurrente.

2.8 DESARROLLO BASADO EN COMPONENTES

Las tecnologías de objetos (4.ª Parte de este libro) proporcionan el marco de trabajo técnico para un modelo de proceso basado en componentes para la ingeniería del software. El paradigma orientado a objetos enfatiza la creación de clases que encapsulan tanto los datos como los algoritmos que se utilizan para manejar los datos. Si se diseñan y se implementan adecuadamente, las clases orientadas a objetos son reutilizables por las diferentes aplicaciones y arquitecturas de sistemas basados en computadora.

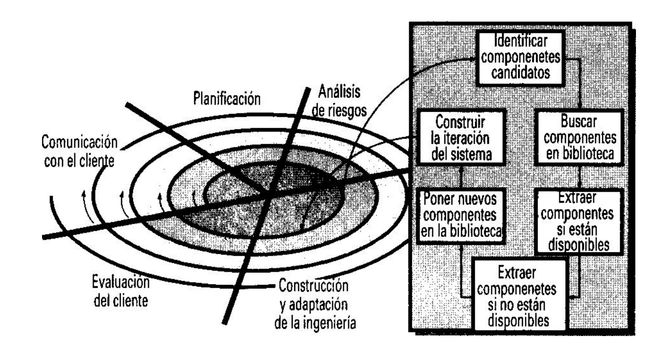


FIGURA 2.11. Desarrollo basado en componentes.

El modelo de desarrollo basado en componentes (Fig. 2.11) incorpora muchas de las características del modelo en espiral. Es evolutivo por naturaleza [NIE92], y exige un enfoque iterativo para la creación del software. Sin embargo, el modelo de desarrollo basado en componentes configura aplicaciones desde componentes preparados de software (llamados «clases» en la Fig. 2.11).

Referencia cruzada

La tecnología resaltada para el DBC se trata en la Parte cuarta de este libro.
En el Capítulo 27 se presenta un estudio más detallado del proceso DBC.

La actividad de la ingeniería comienza con la identificación de clases candidatas. Esto se lleva a cabo examinando los datos que se van a manejar por parte de la aplicación y el algoritmo que se va a aplicar para conseguir el tratamiento[12]. Los datos y los algoritmos correspondientes se empaquetan en una clase.

Las clases creadas en los proyectos de ingeniería del software anteriores, se almacenan en una biblioteca de clases o diccionario de datos (repository) (Capítulo 31). Una vez identificadas las clases candidatas, la biblioteca de clases se examina para determinar si estas clases ya existen. En caso de que así fuera, se extraen de la biblioteca y se vuelven a utilizar. Si una clase candidata no reside en la biblioteca, se aplican los métodos orientados a objetos (Capítulos 21-23). Se compone así la primera iteración de la aplicación a construirse, mediante las clases extraídas de la biblioteca y las clases nuevas construidas para cumplir las necesidades Únicas de la aplicación. El flujo del proceso vuelve a la espiral y volverá a introducir por último la iteración ensambladora de componentes a través de la actividad de ingeniería.

El modelo de desarrollo basado en componentes conduce a la reutilización del software, y la reutilización proporciona beneficios a los ingenieros de software. Según estudios de reutilización, QSM Associates, Inc. informa que el ensamblaje de componentes lleva a una reducción del 70 por 100 de tiempo de ciclo de desarrollo, un 84 por 100 del coste del proyecto y un índice de productividad del 26.2, comparado con la norma de industria del 16.9 [YOU94]. Aunque estos resultados están en función de la robustez de la biblioteca de componentes, no hay duda de que el ensamblaje de componentes proporciona ventajas significativas para los ingenieros de software.

El *proceso unificado de desarrollo de software* [JAC99] representa un número de modelos de desarrollo basados en componentes que han sido propuestos en

[12] Esta es una descripción simplificada de definición de clase. Para un estudio más detallado, consulte el Capítulo 20.

la industria. Utilizando el *Lenguaje de Modelado Unificado* (UML), el proceso unificado define los componentes que se utilizarán para construir el sistema y las interfaces que conectarán los componentes. Utilizando una combinacion del desarrollo incremental e iterativo, el proceso unificado define la función del sistema aplicando un enfoque basado en escenarios (desde el punto de vista del usuario). Entonces acopla la función con un marco de trabajo arquitectónico que identifica la forma que tomará el software.

Referencia cruzada

En los Capítulos **21** y **22** se trata **UML** con más detalle.

2.9. EL MODELO DE MÉTODOS FORMALES

El modelo de métodos formales comprende un conjunto de actividades que conducen a la especificación matemática del software de computadora. Los métodos formales permiten que un ingeniero de software especifique, desarrolle y verifique un sistema basado en computadora aplicando una notación rigurosa y matemática. Algunas organizaciones de desarrollo del software actualmente aplican una variación de este enfoque, llamado *ingeniería del software de sala limpia* [MIL87, DYE92].

Referencia cruzada

los métodos formales se tratan en los Capítulos **25** y 26.

Cuando se utilizan métodos formales (Capítulos 25 y 26) durante el desarrollo, proporcionan un mecanismo para eliminar muchos de los problemas que son difíciles de superar con paradigmas de la ingeniería del software. La ambigüedad, lo incompleto **y** la inconsistencia se descubren y se corrigen más fácilmente —no mediante un revisión a propósito para el caso, sino mediante la aplicación del análisis matemático—. Cuando se utilizan métodos formales durante el diseño, sirven como base'para la verificación de programas y por consiguiente permiten que el ingeniero del software descubra y corrija errores que no se pudieron detectar de otra manera.

Aunque todavía no hay un enfoque establecido, los modelos de métodos formales ofrecen la promesa de un software libre de defectos. Sin embargo, se ha hablado de una gran preocupación sobre su aplicabilidad en un entorno de gestión:

1. El desarrollo de modelos formales actualmente es bastante caro y lleva mucho tiempo.
2. Se requiere un estudio detallado porque pocos responsables del desarrollo de software tienen los antecedentes necesarios para aplicar métodos formales.
3. Es difícil utilizar los modelos como un mecanismo de comunicación con clientes que no tienen muchos conocimientos técnicos.

No obstante es posible que el enfoque a través de métodos formales tenga más partidarios entre los desarrolladores del software que deben construir software de mucha seguridad (por ejemplo: los desarrolladores de aviónica y dispositivos médicos), y entre los desarrolladores que pasan grandes penurias económicas al aparecer errores de software.

2.10 TÉCNICAS DE CUARTA GENERACIÓN

El término técnicas de cuarta generación (T4G) abarca un amplio espectro de herramientas de software que tienen algo en común: todas facilitan al ingeniero del software la especificación de algunas características del software a alto nivel. Luego, la herramienta genera automáticamente el código fuente basándose en la especificación del técnico. Cada vez parece más evidente que cuanto mayor sea el nivel en el que se especifique el software, más rápido se podrá construir el programa. El paradigma T4G para la ingeniería del software se orienta hacia la posibilidad de especificar el software usando formas de lenguaje especializado o notaciones gráficas que describa el problema que hay que resolver en términos que los entienda el cliente. Actualmente, un entorno para el desarrollo de software que soporte el paradigma T4G puede incluir todas o algunas de las siguientes herramientas: lenguajes no procedimentales de consulta a bases de datos, generación de informes, manejo de datos, interacción y definición de pantallas, generación de códigos, capacidades gráficas de alto nivel y capacidades de hoja de cálculo, y generación automatizada de HTML y lenguajes similares utilizados para la creación de sitios web usando herramientas de software avanzado. Inicialmente, muchas de estas herramientas estaban disponibles, pero sólo para ámbitos de aplicación muy específicos, pero actualmente los entornos T4G se han extendido a todas las categorías de aplicaciones del software.

Al igual que otros paradigmas, T4G comienza con el paso de reunión de requisitos. Idealmente, el cliente describe los requisitos, que son, a continuación, traducidos directamente a un prototipo operativo. Sin embar-

go, en la práctica no se puede hacer eso. El cliente puede que no esté seguro de lo que necesita; puede ser ambiguo en la especificación de hechos que le son conocidos, y puede que no sea capaz o no esté dispuesto a especificar la información en la forma en que puede aceptar una herramienta **T4G.** Por esta razón, el diálogo cliente-desarrollador descrito por los otros paradigmas sigue siendo una parte esencial del enfoque **T4G.**



Incluso cuando utilice una T4G, tiene que destacar claramente la ingeniería del software haciendo el análisis, el diseño y las pruebas.

Para aplicaciones pequeñas, se puede ir directamente desde el paso de recolección de requisitos al paso de implementación, usando un lenguaje de cuarta generación no procedimental **(L4G)** o un modelo comprimido de red de iconos gráficos. Sin embargo, es necesario un mayor esfuerzo para desarrollar una estrategia de diseño para el sistema, incluso si se utiliza un **L4G.** El uso de **T4G** sin diseño (para grandes proyectos) causará las mismas dificultades (poca calidad, mantenimiento pobre, mala aceptación por el cliente) que se encuentran cuando se desarrolla software mediante los enfoques convencionales.

La implementación mediante **L4G** permite, al que desarrolla el software, centrarse en la representación de los resultados deseados, que es lo que se traduce automáticamente en un código fuente que produce dichos resultados. Obviamente, debe existir una estructura de datos con información relevante y a la que el **L4G** pueda acceder rápidamente.

Para transformar una implementación **T4G** en un producto, el que lo desarrolla debe dirigir una prueba completa, desarrollar con sentido una documentación y ejecutar el resto de las actividades de integración que son también requeridas requeridas por otros paradigmas de ingeniería del software. Además, el software desarrollado con **T4G** debe ser construido de forma que facilite la realización del mantenimiento de forma expeditiva.

Al igual que todos los paradigmas del software, el modelo **T4G** tiene ventajas e inconvenientes. Los defensores aducen reducciones drásticas en el tiempo de desarrollo del software y una mejora significativa en la productividad de la gente que construye el software. Los detractores aducen que las herramientas actuales de **T4G** no son más fáciles de utilizar que los lenguajes de programación, que el código fuente producido por tales herramientas es «ineficiente» y que el mantenimiento de grandes sistemas de software desarrollados mediante **T4G** es cuestionable.

Hay algún mérito en lo que se refiere a indicaciones de ambos lados y es posible resumir el estado actual de los enfoques de **T4G:**

1. El uso de **T4G** es un enfoque viable para muchas de las diferentes áreas de aplicación. Junto con las herramientas de ingeniería de software asistida por computadora (CASE) y los generadores de código, **T4G** ofrece una solución fiable a muchos problemas del software.
2. Los datos recogidos en compañías que usan **T4G** parecen indicar que el tiempo requerido para producir software se reduce mucho para aplicaciones pequeñas y de tamaño medio, y que la cantidad de análisis y diseño para las aplicaciones pequeñas también se reduce.
3. Sin embargo, el uso de **T4G** para grandes trabajos de desarrollo de software exige el mismo o más tiempo de análisis, diseño y prueba (actividades de ingeniería del software), para lograr un ahorro sustancial de tiempo que puede conseguirse mediante la eliminación de la codificación.

Resumiendo, las técnicas de cuarta generación ya se han convertido en una parte importante del desarrollo de software. Cuando se combinan con enfoques de ensamblaje de componentes (Sección 2.8), el paradigma **T4G** se puede convertir en el enfoque dominante hacia el desarrollo del software.

2.11 TECNOLOGÍAS DE PROCESO

Los modelos de procesos tratados en las secciones anteriores se deben adaptar para utilizarse por el equipo del proyecto del software. Para conseguirlo, se han desarrollado herramientas de tecnología de procesos para ayudar a organizaciones de software a analizar los procesos actuales, organizar tareas de trabajo, controlar y supervisar el progreso y gestionar la calidad técnica [BAN95].

Las herramientas de tecnología de procesos permiten que una organización de software construya un modelo automatizado del marco de trabajo común de proceso, conjuntos de tareas y actividades de protección tratadas en la Sección 2.3. El modelo, presentado normalmente como una red, se puede analizar para determinar el flujo de trabajo típico y para examinar estructuras alternativas de procesos que pudieran llevar a un tiempo o coste de desarrollo reducidos.

Una vez que se ha creado un proceso aceptable, se pueden utilizar otras herramientas de tecnología de procesos para asignar, supervisar e incluso controlar todas las tareas de ingeniería del software definidas como parte del modelo de proceso. Cada uno de los miembros de un equipo de proyecto de software puede utilizar tales

herramientas para desarrollar una lista de control de tareas de trabajo a realizarse, productos de trabajo a producirse, y actividades de garantía de calidad a conducirse. La herramienta de tecnología de proceso también se puede utilizar para coordinar el uso de otras herramientas de ingeniería del software asistida por computadora (Capítulo 31) adecuadas para una tarea de trabajo en particular.

2.12 PRODUCTO Y PROCESO

Si el proceso es débil, el producto final va a sufrir indudablemente. Aunque una dependencia obsesiva en el proceso también es peligrosa. En un breve ensayo, Margaret Davis [DAV95] comenta la dualidad producto y proceso:

Cada diez años o cinco aproximadamente, **la** comunidad del software vuelve a definir «el problema» cambiando el foco de los aspectos de producto a **los** aspectos de proceso. Por consiguiente, se han abarcado lenguajes de programación estructurados (producto) seguidos por métodos de análisis estructurados (proceso) seguidos a su vez por encapsulación de datos (producto) y después por el énfasis actual en el Modelo Madurez de Capacidad de Desarrollo del software del Instituto de ingeniería de software (proceso).

Mientras que la tendencia natural de un péndulo es parar en medio de dos extremos, el enfoque de **la** comunidad del software cambia constantemente porque se aplica una fuerza nueva cuando falla el último movimiento. Estos movimientos son perjudiciales por sí mismos y en **sí** mismos porque confunden al desarrollador del software medio cambiando radicalmente lo que significa realizar el trabajo, que por **sí** mismo lo realiza bien. Los cambios tampoco resuelven «el problema», porque están condenados al fracaso siempre que el producto y el proceso se traten como si formasen una dicotomía en lugar de una dualidad.

Cita:
[Si se desarrolla sin pensar y se aplica descuidadamente, el proceso puede convertirse] en la muerte del sentido común.
Phillip K. Howard

En la comunidad científica hay un precedente que se adelanta a las nociones de dualidad cuando contradicciones en observaciones no se pueden explicar completamente con una teoría competitiva u otra. La naturaleza dual de la luz, que parece ser una partícula y una onda al mismo tiempo, se ha aceptado desde **los** años veinte cuando Louis de Broglie lo propuso. Creo que las observaciones que se hacen sobre **los** mecanismos del software y su desarrollo demuestran una dualidad fundamental entre producto y proceso. Nunca se puede comprender el mecanismo completo, su contexto, uso, significado y valor si se observa **sólo** como un proceso o sólo como un producto...

Toda actividad humana puede ser **un** proceso, pero cada uno de nosotros obtiene el sentido de autoestima ante esas actividades que producen una representación o ejemplo que más de una persona puede utilizar o apreciar, una u otra vez, o en algún otro contexto no tenido en cuenta. Es decir, los sentimientos de satisfacción se obtienen por volver a utilizar nuestros productos por nosotros mismos o por otros.

Así pues, mientras que **la** asimilación rápida de **los** objetivos de reutilización en el desarrollo del software aumenta potencialmente **la** satisfacción que los desarrolladores obtienen de su trabajo, esto incrementa la urgencia de aceptar **la** dualidad producto y proceso. Pensar en un mecanismo reutilizable como el único producto o el único proceso, bien oscurece el contexto y la forma de utilizarlo, o bien el hecho de que cada utilización elabora el producto que a **su** vez se utilizará como entrada en alguna otra actividad de desarrollo del software. Elegir un método sobre otro reduce enormemente las oportunidades de reutilización y de aquí que se pierda la oportunidad de aumentar la satisfacción ante el trabajo.

Cita:
Cualquier actividad se vuelve creativa si el autor se preocupa de hacerlo bien o de hacerlo mejor.
John Updike

Las personas obtienen tanta satisfacción (o más) del proceso creativo que del producto final. Un artista disfruta con las pinceladas de la misma forma que disfruta del resultado enmarcado. Un escritor disfruta con la búsqueda de la metáfora adecuada al igual que con el libro final. Un profesional creativo del software debería también obtener tanta satisfacción de la programación como del producto final.

El trabajo de los profesionales del software cambiará en los próximos años. La dualidad de producto y proceso es un elemento importante para mantener ocupada a la gente creativa hasta que se finalice la transición de la programación a la ingeniería del software.

RESUMEN

La ingeniería del software es una disciplina que integra procesos, métodos y herramientas para el desarrollo del software de computadora. Se han propuesto varios modelos de procesos para la ingeniería del software diferentes, cada uno exhibiendo ventajas e inconvenientes, pero todos tienen una serie de fases genéricas en común. En el resto de este libro se consideran los principios, conceptos y métodos que permiten llevar a cabo el proceso que se llama ingeniería del software.

REFERENCIAS

[BAE98] Baetjer, H., Jr., *Software as Capital,* IEEE Computer Society Press, 1998, p. 85.

[BAN95] Bandinelli, S. et al, «Modeling and Improving an Industrial Software Process», IEEE *Trans. Software Engineering,* vol. 21, n.º 5, Mayo 1995, pp. 440-454.

[BRA94] Bradac, M., D. Perry y L. Votta, «Prototyping a Process Monitoring Experiment", IEEE *Trans. Software Engineering,* vol. 20, n.º 10, Octubre 1994, pp. 774-784.

[BOE88] Boehm, B., «A Spiral Model for Software Developement and Enhancement", *Computer,* vol. 21, n.º 5, Mayo 1988, pp. 61-72.

[BOE96] Boehm, B., «Anchoring the Software Pocess», *IEEE Software,* vol. 13, n.º 4, Julio 1996, pp. 73-82.

[BOE98] Boehm, B., «Using the WINWIN Spiral Model: A Case Study», *Computer,* vol. 31, n.º 7, Julio 1998, pp. 33-44.

[BRO75] Brooks, F., *The Mytical Man-Month,* Addison-Wesley, 1975.

[BUT94] Butler, J., «Rapid Aplication Developement in Action», *Managing System Developement,* Applied Computer Research, vol. 14, n.º 5, Mayo 1995, pp. 6-8.

[DAV94] Davis, A., y P. Sitaram, «A Concurrent Process Model for Software Developement», *Software Engineering Notes,* ACM Press, vol. 19, n.º 2, Abril 1994, pp. 38-51.

[DAV95] Davis, M.J., «Process and Product: Dichotomy or Duality», *Software Engineering Notes,* ACM Press, vol. 20, n.º 2, Abril 1995, pp. 17-18.

[DON96] Donaldson, M.C., y M. Donaldson, *Negotiating for Dummies,* IDB Books Worldwide, 1996.

[DYE92] Dyer, M., *The Cleanroom Approach to Quality Software Developement,* Wiley, 1992.

[FAR97] Farber, D.C., *Common Sense Negotiation: The Art of Winning Gracefully,* Bay Press, 1997.

[FIS91] Fisher, R., W. Ury y Bruce Patton, *Getting to Yes: Negotiating Agreement Without Giving In,* 2." ed., Penguin USA, 1991.

[GIL88] Gilb, T., *Principles of Software Engineering Management,* Addison-Wesley, 1988.

[HAN95] Hanna, M., «Farewell to Waterfalls», *Software Magazine,* Mayo 1995, pp. 38-46.

[HUM89] Humphrey, W., y M. Kellner, «Software Process Modeling: Principles of Entity Process Models», *Proc. 11th Intl. Conference on Software Engineering,* IEEE Computer Society Press, pp. 331-342.

[IEE93] IEEE *Standards Collection: Software Engineering,* IEEE Standard 610.12-1990, IEEE, 1993.

[JAC99] Jacobson, I., G. Booch y J. Rumbaugh, *The Unified Software Developement Proccess,* Addison-Wesley, 1999.

[KEL89] Kellner, M., *Software Process Modeling: Value and Experience,* SEI Technical Review- 1989, SEI, Pittsburgh, PA, 1989.

[KEL91] Kellner, M., «Software Process Modeling Support for Management Planning and Control», *Proc. 1st Intl. Conf, On the Software Process,* IEEE Computer Society Press, 1991, pp. 8-28.

[KER94] Kerr, J., y R. Hunter, *Inside RAD,* McGraw-Hill, 1994.

[MAR91] Martin, J., *Rapid Aplication Developement,* Prentice-Hall, 1991.

[MDE93] McDermid, J. y P. Rook, «Software Developement Process Models», en *Software Ingineer's Reference Book,* CRC Press, 1993, pp. 15/26-15/28.

[MIL87] Mills, H.D., M. Dyer y R. Linger, «Clearoom Software Engineering», *IEEE Software,* Septiembre 1987, pp. 19-25.

[NAU69] Naur, P., y B. Randall (eds.), *Software Engineering: A Report on a Conference sponsored by the NATO Science Committee,* NATO, 1969.

[NIE92] Nierstrasz, «Component-Oriented Software Developement», CACM, vol. 35, n.º 9, Septiembre 1992, pp. 160-165.

[PAU93] Paulk, M., et al., «Capability Maturity Model for Software», *Software Engineering Institute,* Carnegie Mellon University, Pittsburgh, PA, 1993.

[RAC95] Raccoon, L.B.S, «The Chaos Model and the Chaos Life Cycle», *ACM Software Engineering Notes,* vol. 20, n.º 1, Enero 1995, pp. 55-66.

[ROY70] Royce, W.W., «Managing the Developement of Large Software Systems: Concepts and Techniques», *Proc. WESCON,* Agosto 1970.

[SHE94] Sheleg, W., «Concurrent Engineering: A New Paradign for C/S Developement», *Application Developement Trends,* vol. 1, n.º 6, Junio 1994, pp. 28-33.

[YOU94] Yourdon, E., «Software Reuse», *Application Developement Strategies,* vol. VI, n.º 12, Diciembre 1994, pp. 1-16.

PROBLEMAS Y PUNTOS A CONSIDERAR

2.1. La Figura 2.1 sitúa las tres capas de ingeniería del software encima de la capa titulada «enfoque de calidad». Esto implica un programa de calidad tal como Gestión de Calidad Total. Investigue y desarrolle un esquema de los principios clave de un programa de Gestión de Calidad Total.

2.2. ¿Hay algún caso en que no se apliquen fases genéricas del proceso de ingeniería del software? Si es así, descríbalo.

2.3. El Modelo Avanzado de Capacidad del SEI es un documento en evolución. Investigue y determine si se han añadido algunas ACP nuevas desde la publicación de este libro.

2.4. El modelo del caos sugiere que un bucle de resolución de problemas se pueda aplicar en cualquier grado de resolución. Estudie la forma en que se aplicaría el bucle (1) para comprender los requisitos de un producto de tratamiento de texto; **(2)** para desarrollar un componente de corrección ortográfica y gramática avanzado para el procesador de texto; (3) para generar el código para un módulo de programa que determine el sujeto, predicado y objeto de una oración en inglés.

2.5. ¿**Qué** paradigmas de ingeniería del software de los presentados en este capítulo piensa que sería el más eficaz? ¿**Por** qué?

2.6. Proporcione cinco ejemplos de proyectos de desarrollo del software que sean adecuados para construir prototipos. Nombre dos o tres aplicaciones que fueran más difíciles para construir prototipos.

2.7. El modelo DRA a menudo se une a herramientas CASE. Investigue la literatura y proporcione un resumen de una herramienta típica CASE que soporte DRA.

2.8. Proponga un proyecto específico de software que sea adecuado para el modelo incremental. Presente un escenario para aplicar el modelo al software.

2.9. A medida que vaya hacia afuera por el modelo en espiral, ¿**qué** puede decir del software que se está desarrollando o manteniendo?

2.10. Muchas personas creen que la Única forma en la que se van a lograr mejoras de magnitud en la calidad del software y en su productividad es a través del desarrollo basado en componentes. Encuentre tres o cuatro artículos recientes sobre el asunto y resúmalos para la clase.

2.11. Describa el modelo de desarrollo concurrente con sus propias palabras.

2.12. Proporcione tres ejemplos de técnicas de cuarta generación.

2.13. ¿Qué es más importante, el producto o el proceso?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

El estado actual del arte en la ingeniería del software se puede determinar mejor a partir de publicaciones mensuales tales como ***IEEE Software, Computer e IEEE Transactions** on **Software Engineering.*** Periódicos sobre la industria tales como *Application Developement Trends, Cutter IT Journal* y ***Software** Developement* a menudo contienen artículos sobre temas de ingeniería del software. La disciplina se «resume» cada año en el *Proceeding of the International Conference on Software Engineering,* promocionado por el IEEE y ACM **y** tratado en profundidad en periódicos como ***ACM Transactions** on **Software Engineering and Methodology, ACM Software Engineering Notes*** y *Annals of Software Engineering.*

Se han publicado muchos libros de ingeniería del software **en** los últimos años. Algunos presentan una visión general del proceso entero mientras que otros se dedican a unos pocos temas importantes para la exclusión de otros. Las siguientes son tres antologías que abarcan algunos temas importantes de ingeniería del software:

Keyes, **J.** (ed.), *Software **Engineering Productivity** Handbook,* McGraw-Hill, 1993.

McDermid, **J.** (ed.), ***Software** Engineer's **Reference Book,*** **CRC** Press, 1993.

Marchiniak, **J.J.** (ed.), ***Encyclopedia** of **Software** Engineering,* Wiley, 1994.

Gautier ***(Distributed Engineering** of **Software,*** Prentice-Hall, 1996) proporciona sugerencias y directrices para organizaciones que deban desarrollar software en lugares geográficamente distantes.

En la parte más brillante del libro de Robert Glass ***(Software** Conflict,* Yourdon Press, 1991) se presentan ensayos controvertidos y amenos sobre el software y el proceso de la ingeniería del software. Pressman y Herron ***(Software Shock,*** Dorset House, 1991) consideran el software y su impacto sobre particulares, empresas y el gobierno.

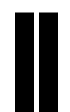
El Instituto de ingeniería del software (IIS localizado en la Universidad de Carnegie-Mellon) ha sido creado con la responsabilidad de promocionar series monográficas sobre la ingeniería del software. Los profesionales académicos, en la industria y el gobierno están contribuyendo con nuevos trabajos importantes. El Consorcio de Productividad del Software dirige una investigación adicional de ingeniería de software.

En la última década se ha publicado una gran variedad de estándares y procedimientos de ingeniería del software. El ***IEEE Software Engineering Standards*** contiene muchos estándares diferentes que abarcan casi todos los aspectos importantes de la tecnología. Las directrices ISO 9000-3 proporcionan una guía para las organizaciones de software que requieren un registro en el estándar de calidad ISO 9001. Otros estándares de ingeniería del software se pueden obtener desde el MOD, la Autoridad Civil de Aviación y otras agencias del gobierno y sin ánimo de lucro. Fairclough ***(Software Engineering Guides,*** Prentice-Hall, 1996) proporciona una referencia detallada de los estándares de ingeniería del software producidos por European Space Agency (ESA).

En internet se dispone de una gran variedad de fuentes de información sobre la ingeniería del software y del proceso de software. Se puede encontrar una lista actualizada con referencias a sitios (páginas) web que son relevantes para el proceso del software en **http://www.pressman5.com.**

.

PARTE



GESTION DE PROYECTOS DE SOFTWARE

EN esta parte de *Ingeniería del Software: Un enfoque Práctico,* estudiamos las técnicas de gestión necesarias para planificar, organizar, supervisar y controlar proyectos de software. En los capítulos que vienen a continuación, obtendremos respuestas a las siguientes cuestiones:

- ¿Cómo se debe gestionar el personal, el proceso y el problema durante un proyecto de software?
- ¿Qué son las métricas de software y cómo pueden emplearse para gestionar un proyecto de software y el proceso del software?
- ¿Cómo genera un equipo de software estimaciones fiables del esfuerzo, costes y duración del proyecto?
- ¿Qué técnicas pueden emplearse para evaluar formalmente los riesgos que pueden tener impacto en el éxito del proyecto?
- ¿Cómo selecciona un gestor de proyectos de software el conjunto de tareas del proyecto de ingeniería de software?
- ¿Cómo se crea la planificación temporal de un proyecto?
- ¿Cómo se define la calidad para que pueda ser controlada?
- ¿Qué es la garantía de calidad del software?
- ¿Por qué son tan importantes las revisiones técnicas formales?
- ¿Cómo se gestionan los cambios durante el desarrollo de software de computadora y después de entregarlo al cliente?

Una vez que se haya dado respuesta a estas cuestiones, estará mejor preparado para gestionar proyectos de software de manera que entregue un producto de alta calidad puntualmente.

CAPÍTULO

3 CONCEPTOS SOBRE GESTION DE PROYECTOS

EN el prefacio de su libro sobre gestión de proyectos de software, Meiler Page-Jones [PAG85] dice una frase que pueden corroborar muchos asesores de ingeniería del software:

> He visitado docenas de empresas, buenas y malas, y he observado a muchos administradores de proceso de datos, también buenos y malos. Frecuentemente, he visto con horror cómo estos administradores luchaban inútilmente contra proyectos de pesadilla, sufriendo por fechas límite imposibles de cumplir, o que entregaban sistemas que decepcionaban a sus usuarios y que devoraban ingentes cantidades de horas de mantenimiento.

Lo que describe Page-Jones son síntomas que resultan de una serie de problemas técnicos y de gestión. Sin embargo, si se hiciera una autopsia de cada proyecto, es muy probable que se encontrara un denominador común constante: una débil gestión.

En este capítulo y en los seis que siguen consideraremos los conceptos clave que llevan a una gestión efectiva de proyectos de software. Este capítulo trata conceptos y principios básicos sobre gestión de proyectos de software. El Capítulo 4 presenta las métricas del proyecto y del proceso, la base para una toma de decisiones de gestión efectivas. Las técnicas que se emplean para estimar los costes y requisitos de recursos y poder establecer un plan efectivo del proyecto se estudian en el Capítulo 5. Las actividades de gestión que llevan a una correcta supervisión, reducción y gestión del riesgo se presentan en el Capítulo 6. El Capítulo 7 estudia las actividades necesarias para definir las tareas de un proyecto y establecer una programación del proyecto realista. Finalmente, los Capítulos 8 y 9 consideran técnicas para asegurar la calidad a medida que se dirige un proyecto y el control de los cambios a lo largo de la vida de una aplicación.

VISTAZO RÁPIDO

¿Qué es? Aunque muchos de nosotros (en nuestros momentos más oscuros) tomamos la visión de Dilbert sobre la «gestión», falta una actividad muy necesaria cuando se construyen productos y sistemas basados en computadoras. La gestión de proyectos implica la planificación, supervisión y control del personal, del proceso y de los eventos que ocurren mientras evoluciona el software desde la fase preliminar a la implementación operacional.

¿Quién lo hace? Todos «gestionamos» de algún modo, pero el ámbito de las actividades de gestión varía en función de la persona que las realiza. Un ingeniero de software gestiona sus actividades del día a día, planificando, supervisando, y controlando las tareas técnicas. Los gestores del proyecto planifican, supervisan y controlan el trabajo de un equipo de ingenieros de software. Los gestores expertos coordinan la relación entre el negocio y los profesionales del software.

¿Por qué es importante? La construcción de software de computadora es una empresa compleja, particularmente si participa mucha gente, trabajando durante un periodo de tiempo relativamente largo. Esta es la razón por la cual los proyectos de software necesitan ser gestionados.

¿Cuáles son los pasos? Comprender las cuatro «P's» —personal, producto, proceso y proyecto—. El personal debe estar organizado para desarrollar el trabajo del software con efectividad. La comunicación con el cliente debe ocurrir para que se comprendan el alcance del producto y los requisitos. Debe seleccionarse el proceso adecuado para el personal, y el producto. El proyecto debe planificarse estimando el esfuerzo y el tiempo para cumplir las tareas; definiendo los productos del trabajo, estableciendo puntos de control de calidad y estableciendo mecanismos para controlar y supervisar el trabajo definido en la planificación.

¿Cuál es el producto obtenido? Un plan de proyecto se realiza al comienzo de las actividades de gestión. El plan define el proceso y las tareas a realizar el personal que realizará el trabajo y los mecanismos para evaluar los riesgos, controlar el cambio y evaluar la calidad.

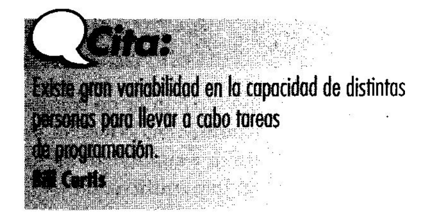
¿Cómo puedo estar seguro de que lo he hecho correctamente? Nunca estás completamente seguro de que el plan de proyecto es correcto hasta que no has entregado un producto de alta calidad dentro del tiempo y del presupuesto. Sin embargo, un gestor de proyectos hace lo correcto cuando estimula al personal para trabajar juntos como un equipo efectivo, centrando su atención en las necesidades del cliente y en la calidad del producto.

3.1 EL ESPECTRO DE LA GESTIÓN

La gestión eficaz de un proyecto de software se centra en las cuatro P's: *personal, producto, proceso y proyecto*. El orden no es arbitrario. El gestor que se olvida de que el trabajo de ingeniería del software es un esfuerzo humano intenso nunca tendrá éxito en la gestión de proyectos. Un gestor que no fomenta una minuciosa comunicación con el cliente al principio de la evolución del proyecto se arriesga a construir una elegante solución para un problema equivocado. El administrador que presta poca atención al proceso corre el riesgo de arrojar métodos técnicos y herramientas eficaces al vacío. El gestor que emprende un proyecto sin un plan sólido arriesga el éxito del producto.

3.1.1. Personal

La necesidad de contar con personal para el desarrollo del software altamente preparado y motivado se viene discutiendo desde los años 60 (por ejemplo, [COU80, WIT94, DEM98]). De hecho, el «factor humano» es tan importante que el Instituto de Ingeniería del Software ha desarrollado un *Modelo de madurez de la capacidad de gestión de personal* (MMCGP) «para aumentar la preparación de organizaciones del software para llevar a cabo las cada vez más complicadas aplicaciones ayudando a atraer, aumentar, motivar, desplegar **y** retener el talento necesario para mejorar su capacidad de desarrollo de software» [CUR94].



El modelo de madurez de gestión de personal define las siguientes áreas clave prácticas para el personal que desarrolla software: reclutamiento, selección, gestión de rendimiento, entrenamiento, retribución, desarrollo de la carrera, diseño de la organización **y** del trabajo y desarrollo cultural y de espíritu de equipo.

El MMCGP es compañero del modelo de madurez de la capacidad software (Capítulo 2), que guía a las organizaciones en la creación de un proceso de software maduro. **Más** adelante en este capítulo se consideran aspectos asociados a la gestión de personal y estructuras para proyectos de software.

3.1.2. Producto

Antes de poder planificar un proyecto, se deberían establecer los objetivos y el ámbito del producto[1], se deberían considerar soluciones alternativas e identificar las dificultades técnicas y de gestión. Sin esta información, es imposible definir unas estimaciones razonables (y exactas) del coste; una valoración efectiva del riesgo, una subdivisión realista de las tareas del proyecto o una planificación del proyecto asequible que proporcione una indicación fiable del progreso.

Referencia cruzada
En el Capítulo 1 se trata una taxonomía de las áreas de aplicación que producen los «productos» de software.

El desarrollador de software **y** el cliente deben reunirse para definir los objetivos del producto y su ámbito. En muchos casos, esta actividad empieza como parte del proceso de ingeniería del sistema o del negocio (Capítulo 10) y continúa como el primer paso en el análisis de los requisitos del software (Capítulo 11). Los objetivos identifican las metas generales del proyecto sin considerar cómo se conseguirán (desde el punto de vista del cliente).

El ámbito identifica los datos primarios, funciones y comportamientos que caracterizan al producto, y, más importante, intenta *abordar* estas características de una manera cuantitativa.

Una vez que se han entendido los objetivos y el ámbito del producto, se consideran soluciones alternativas.

3.1.3. Proceso

Un proceso de software (Capítulo 2) proporciona la estructura desde la que se puede establecer un detallado plan para el desarrollo del software. Un pequeño número de actividades estructurales se puede aplicar a todos los proyectos de software, sin tener en cuenta su tamaño o complejidad. Diferentes conjuntos de tareas —tareas, hitos, productos del trabajo y puntos de garantía de calidad— permiten a las actividades estructurales adaptarse a las características del proyecto de software y a los requisitos del equipo del proyecto. Finalmente, las actividades protectoras —tales como garantía de calidad del software, gestión de la configuración del software y medición— cubren el modelo de proceso. Las actividades protectoras son independientes de las estructurales y tienen lugar a lo largo del proceso.

[1] En este contexto, el termino «producto» es usado para abarcar cualquier software que será construido a petición de otros. Esto incluye no sólo productos de software, sino también sistemas basados en computadora, software empotrado y software de resolución de problemas (por ejemplo, programas para la resolución de problemas científicos/de ingeniería).

PUNTO CLAVE

las actividades estructurales se componen de tareas, hitos, productos de trabajo y puntos de garantía de calidad.

3.1.4. Proyecto

Dirigimos los proyectos de software planificados y controlados por una razón principal -es la Única manera conocida de gestionar la complejidad—. Y todavía seguimos esforzándonos. En 1998, los datos de la industria del software indicaron que el 26 por 100 de proyectos de software fallaron completamente y que el 46 por 100 experimentaron un desbordamiento en la planificación y en el coste [REE99]. Aunque la proporción de éxito para los proyectos de software ha mejorado un poco, nuestra proporción de fracaso de proyecto permanece más alto del que debería ser[2].

Para evitar el fracaso del proyecto, un gestor de proyectos de software y los ingenieros de software que construyeron el producto deben eludir un conjunto de señales de peligro comunes; comprender los factores del éxito críticos que conducen a la gestión correcta del proyecto y desarrollar un enfoque de sentido común para planificar, supervisar y controlar el proyecto. Cada uno de estos aspectos se trata en la Sección 3.5 y en los capítulos siguientes.

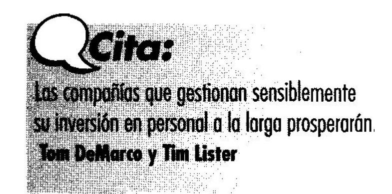
3.2 PERSONAL

En un estudio publicado por el IEEE [CUR88] se les preguntó a los vicepresidentes ingenieros de tres grandes compañías tecnológicas sobre el factor más importante que contribuye al éxito de un proyecto de software. Respondieron de la siguiente manera:

VP 1: Supongo que si tuviera que elegir lo más importante de nuestro entorno de trabajo, diría que no son las herramientas que empleamos, es la gente.

VP 2: El ingrediente más importante que contribuyó **al éxi**to de este proyecto fue tener gente lista **...**pocas cosas más importan en mi opinión **...** Lo más importante que se puede hacer por un proyecto es seleccionar el personal **...** El éxito de la organización de desarrollo del software está muy, muy asociado con la habilidad de reclutar buenos profesionales.

VP **3:** La única regla que tengo en cuanto a la gestión es asegurarme de que tengo buenos profesionales —gente realmente buena—, de que preparo buena gente y de que proporciono el entorno en el que los buenos profesionales puedan producir.



Ciertamente, éste es un testimonio convincente sobre la importancia del personal en el proceso de ingeniería del software. Y, sin embargo, todos nosotros, desde los veteranos vicepresidentes al más modesto profesional del software, damos este aspecto por descontado. Los gestores argumentan (como el grupo anterior) que el personal es algo primario, pero los hechos desmienten a veces sus palabras.

En esta sección examinamos los participantes que colaboran en el proceso del software y la manera en que se organizan para realizar una ingeniería del Software eficaz.

3.2.1. Los participantes

El proceso del software (y todos los proyectos de software) lo componen participantes que pueden clasificarse en una de estas cinco categorías:

1. *Gestores superiores*, que definen los aspectos de negocios que a menudo tienen una significativa influencia en el proyecto.
2. *Gestores (técnicos) del proyecto*, que deben planificar, motivar, organizar y controlar a los profesionales que realizan el trabajo de software.
3. *Profesionales*, que proporcionan las capacidades técnicas necesarias para la ingeniería de un producto o aplicación.
4. *Clientes*, que especifican los requisitos para la ingeniería del software y otros elementos que tienen menor influencia en el resultado.
5. *Usuarios finales*, que interaccionan con el software una vez que se ha entregado para la producción.

Para ser eficaz, el equipo del proyecto debe organizarse de manera que maximice las habilidades y capacidades de cada persona. Y este es el trabajo del jefe del equipo.

[2] Dadas estas estadísticas, es razonable preguntarse cómo el impacto de las computadoras continúa creciendo exponencialmente y la industria del software continúa anunciando el crecimiento de ventas al doble. Parte de la respuesta, pienso, es que un importante número de estos proyectos «fallidos» están mal concebidos desde el primer momento. Los clientes pierden el interés rápidamente (puesto que lo que ellos pidieron realmente no era tan importante como ellos habían pensado), y los proyectos son cancelados.

3.2.2. Los jefes de equipo

La gestión de un proyecto es una actividad intensamente humana, y por esta razón, los profesionales competentes del software a menudo no son buenos jefes de equipo. Simplemente no tienen la mezcla adecuada de capacidades del personal. Y sin embargo, como dice Edgemon: «Desafortunadamente y con demasiada frecuencia, hay individuos que terminan en la gestión de proyectos y se convierten en gestores de proyecto accidentales» [EDG95].

¿En qué nos fijamos cuando seleccionamos a alguien para conducir un proyecto de software?

En un excelente libro sobre gestión técnica, Jerry Weinberg [WEI86] sugiere el modelo de gestión MOI:

Motivación. La habilidad para motivar (con un «tira y afloja») al personal técnico para que produzca conforme a sus mejores capacidades.

Organización. La habilidad para amoldar procesos existentes (o inventar unos nuevos) que permita al concepto inicial transformarse en un producto final.

Ideas o innovación. La habilidad para motivar al personal para crear y sentirse creativos incluso cuando deban de trabajar dentro de los límites establecidos para un producto o aplicación de software particular.

Cita:

En equipos sencillos un líder es aquel que sabe a donde quiere ir, se levanta y va.

John Erskine

Weinberg sugiere que el éxito de los gestores de proyecto se basa en aplicar un estilo de gestión en la resolución de problemas. Es decir, un gestor de proyectos de software debería concentrarse en entender el problema que hay que resolver, gestionando el flujo de ideas y, al mismo tiempo, haciendo saber a todos los miembros del equipo (mediante palabras y, mucho más importante, con hechos) que la calidad es importante y que no debe verse comprometida.

Otro punto de vista [EDG95] de las características que definen a un gestor de proyectos eficiente resalta cuatro apartados clave:

Resolución del problema. Un gestor eficiente de un proyecto de software puede diagnosticar los aspectos técnicos y de organización más relevantes, estructurar una solución sistemáticamente o motivar apropiadamente a otros profesionales para que desarrollen la solución, aplicar las lecciones aprendidas de anteriores proyectos a las nuevas situaciones, mantenerse lo suficientemente flexible para cambiar la gestión si los intentos iniciales de resolver el problema no dan resultado.

Dotes de gestión. Un buen gestor de proyectos debe tomar las riendas. Debe tener confianza para asumir el control cuando sea necesario y la garantía para permitir que los buenos técnicos sigan sus instintos.

Incentivos por logros. Para optimizar la productividad de un equipo de proyecto, un gestor debe recompensar la iniciativa y los logros, y demostrar a través de sus propias acciones que no se penalizará si se corren riesgos controlados.

Influencia y construcción de espíritu de equipo. Un gestor de proyecto eficiente debe ser capaz de «leer» a la gente; debe ser capaz de entender señales verbales y no verbales y reaccionar ante las necesidades de las personas que mandan esas señales. El gestor debe mantener el control en situaciones de gran estrés.



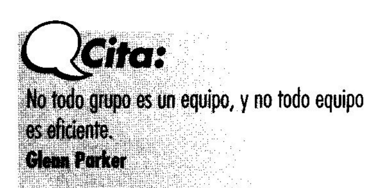
Un experto de software puede no tener temperamento o deseo de ser jefe de equipo. No fuerce al experto poro ser uno de ellos.

3.2.3. El equipo de software

Existen casi tantas estructuras de organización de personal para el desarrollo de software como organizaciones que se dedican a ello. Para bien o para mal, el organigrama no puede cambiarse fácilmente. Las consecuencias prácticas y políticas de un cambio de organización no están dentro del alcance de las responsabilidades del gestor de un proyecto de software. Sin embargo, la organización del personal directamente involucrado en un nuevo proyecto de software está dentro del ámbito del gestor del proyecto.

Las siguientes opciones pueden aplicarse a los recursos humanos de un proyecto que requiere n personas trabajando durante k años:

1. *n* individuos son asignados a *m* diferentes tareas funcionales, tiene lugar relativamente poco trabajo conjunto; la coordinación es responsabilidad del gestor del software que puede que tenga otros seis proyectos de los que preocuparse.



2. *n* individuos son asignados a m diferentes tareas funcionales ($m<n$) de manera que se establecen «equipos» informales; se puede nombrar un líder al efecto; la coordinación entre los equipos es responsabilidad de un gestor del software.
3. *n* individuos se organizan en *t* equipos; a cada equipo se le asignan una o más tareas funcionales; cada equipo tiene una estructura específica que se define para todos los equipos que trabajan en el proyecto; la coordinación es controlada por el equipo y por el gestor del proyecto de software.

Aunque es posible encontrar argumentos en pro y en contra para cada uno de los enfoques anteriores, existe

una gran evidencia que indica que una organización de equipo formal (opción 3) es la más productiva.

La «mejor» estructura de equipo depende del estilo de gestión de una organización, el número de personas que compondrá el equipo, sus niveles de preparación y la dificultad general del problema. Mantei [MAN81] sugiere tres organigramas de equipo genéricos:

> **Descentralizado democrático (DD).** Este equipo de ingeniería del software no tiene un jefe permanente. **Más** bien, «se nombran coordinadores de tareas a corto plazo y se sustituyen por otros para diferentes tareas». Las decisiones sobre problemas y los enfoques se hacen por consenso del grupo. La comunicación entre los miembros del equipo es horizontal.

¿Cómo debería estar organizado un equipo de software?

> **Descentralizado controlado (DC).** Este equipo de ingeniería del software tiene un jefe definido que coordina tareas específicas y jefes secundarios que tienen responsabilidades sobre subtareas. La resolución de problemas sigue siendo una actividad del grupo, pero la implementación de soluciones se reparte entre subgrupos por el jefe de equipo. La comunicación entre subgrupos e individuos es horizontal. También hay comunicación vertical a lo largo de la jerarquía de control.

> **Centralizado controlado (CC).** El jefe del equipo se encarga de la resolución de problemas a alto nivel y la coordinación interna del equipo. La comunicación entre el jefe y los miembros del equipo es vertical.

Mantei [MAN81] describe siete factores de un proyecto que deberían considerarse cuando se planifica el organigrama de equipos de ingeniería del software:

- la dificultad del problema que hay que resolver
- el tamaño del programa(s) resultante(s) en líneas de código o puntos de función (Capítulo 4)
- el tiempo que el equipo estará junto (tiempo de vida del equipo)
- el grado en que el problema puede ser modularizado

¿Qué factores deberíamos considerar cuando estructuramos un equipo de software?

- la calidad requerida y fiabilidad del sistema que se va a construir
- la rigidez de la fecha de entrega
- el grado de sociabilidad (comunicación) requerido para el proyecto

Debido a que una estructura centralizada realiza las tareas más rápidamente, es la más adecuada para manejar problemas sencillos. Los equipos descentralizados generan más y mejores soluciones que los individuales. Por tanto, estos equipos tienen más probabilidades de éxito en la resolución de problemas complejos. Puesto que el equipo DC es centralizado para la resolución de problemas, tanto el organigrama de equipo DC como el de CC pueden aplicarse con éxito para problemas sencillos. La estructura DD es la mejor para problemas difíciles.

Como el rendimiento de un equipo es inversamente proporcional a la cantidad de comunicación que se debe entablar, los proyectos muy grandes son mejor dirigidos por equipos con estructura CC o DC, donde se pueden formar fácilmente subgrupos.

El tiempo que los miembros del equipo vayan a «vivir juntos» afecta a la moral del equipo. Se ha descubierto que los organigramas de equipo tipo DD producen una moral más alta y más satisfacción por el trabajo y son, por tanto, buenos para equipos que permanecerán juntos durante mucho tiempo.

El organigrama de equipo DD se aplica mejor a problemas con modularidad relativamente baja, debido a la gran cantidad de comunicación que se necesita. Los organigramas CC o DC funcionan bien cuando es posible una modularidad alta (y la gente puede hacer cada uno lo suyo).



Frecuentemente es mejor tener pocos equipos pequeños, bien centrados que un gran equipo solamente.

Los equipos CC y DC producen menos defectos que los equipos DD, pero estos datos tienen mucho que ver con las actividades específicas de garantía de calidad que aplica el equipo. Los equipos descentralizados requieren generalmente más tiempo para completar un proyecto que un organigrama centralizado y al mismo tiempo son mejores cuando se precisa una gran cantidad de comunicación.

Constantine [CON93] sugiere cuatro «paradigmas de organización» para equipos de ingeniería del software:

1. *Un paradigma cerrado* estructura a un equipo con una jerarquía tradicional de autoridad (similar al equipo CC). Estos equipos trabajan bien cuando producen software similar a otros anteriores, pero probablemente sean menos innovadores cuando trabajen dentro de un paradigma cerrado.
2. *El paradigma aleatorio* estructura al equipo libremente y depende de la iniciativa individual de los miembros del equipo. Cuando se requiere innovación o avances tecnológicos, los equipos de paradigma aleatorio son excelentes. Pero estos equipos pueden chocar cuando se requiere un «rendimiento ordenado».

3. *El paradigma abierto* intenta estructurar a un equipo de manera que consiga algunos de los controles asociados con el paradigma cerrado, pero también mucha de la innovación que tiene lugar cuando se utiliza el paradigma aleatorio. El trabajo se desarrolla en colaboración, con mucha comunicación y toma de decisiones consensuadas y con el sello de los equipos de paradigma abierto. Los organigramas de equipo de paradigma abierto son adecuados para la resolución de problemas complejos, pero pueden no tener un rendimiento tan eficiente como otros equipos.

Cita:
Trabajar en equipo es difícil, pero no imposible.
Peter Drucker

4. El *paradigma sincronizado* se basa en la compartimentación natural de un problema y organiza los miembros del equipo para trabajar en partes del problema con poca comunicación activa entre ellos.

Constantine [CON93] propone una variación en el equipo descentralizado democrático defendiendo a los equipos con independencia creativa cuyo enfoque de trabajo podría ser mejor llamado «anarquía innovadora». Aunque se haya apelado al enfoque de libre espíritu para el desarrollo del software, el objetivo principal de una organización de Ingeniería del Software debe ser «convertir el caos en un equipo de alto rendimiento» [HYM93]. Para conseguir un equipo de alto rendimiento.

- Los miembros del equipo deben confiar unos en otros.
- La distribución de habilidades debe adecuarse al problema.
- Para mantener la unión del equipo, los inconformistas tienen que ser excluidos del mismo

Cualquiera que sea la organización del equipo, el objetivo para todos los gestores de proyecto es colaborar a crear un equipo que presente cohesión. En su libro, *Peopfeware*, DeMarco y Lister [DEM98] estudian este aspecto:

Referencia cruzada
El papel del **bibliotecario** existe sin tener en cuento lo estructura del equipo. Poro **más** detalles **véase** el **Capítulo 9.**

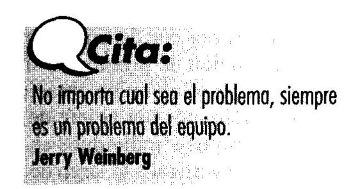
Tendemos a usar la palabra equipo demasiado libremente en el mundo de los negocios, denominando «equipo» a cualquier grupo de gente asignado para trabajar junta. Pero muchos de estos grupos no parecen equipos. No tienen una definición común de éxito o un espíritu de equipo identificable. Lo que falta es un fenómeno que denominamos *cuajar.*

Un equipo cuajado es un grupo de gente tejido tan fuertemente que el todo es mayor que la suma de las partes ...

Una vez que el equipo empieza a cuajar, la probabilidad de éxito empieza a subir. El equipo puede convertirse en imparable, un monstruo de éxito ... No necesitan ser dirigidos de una manera tradicional y no necesitan que se les motive. Están en su gran momento.

DeMarco y Lister mantienen que los miembros de equipos cuajados son significativamente más productivos y están más motivados que la media. Comparten una meta común, una cultura común y, en muchos casos, un «sentimiento elitista» que les hace únicos.

Pero no todos los equipos cuajan. De hecho, muchos equipos sufren lo que Jackman llama «toxicidadde equipo» [JAC98] define cinco factores que «fomentan un entorno de equipo tóxico potencial»:



1. una atmósfera de trabajo frenética en la que los miembros del equipo gastan energía y se descentran de los objetivos del trabajo a desarrollar;
2. alta frustración causada por factores tecnológicos, del negocio, o personales que provocan fricción entre los miembros del equipo;
3. «procedimientos coordinados pobremente o fragmentados» o una definición pobre o impropiamente elegida del modelo de procesos que se convierte en un obstáculo a saltar;
4. definición confusa de los papeles a desempeñar produciendo una falta de responsabilidad y la acusación correspondiente, y
5. «continua y repetida exposición al fallo» que conduce a una pérdida de confianza y a una caída de la moral.

Jackman sugiere varios antitóxicos que tratan los problemas comunes destacados anteriormente.

Para evitar un entorno de trabajo frenético, el gestor de proyectos debería estar seguro de que el equipo tiene acceso a toda la información requerida para hacer el trabajo y que los objetivos y metas principales, una vez definidos, no deberían modificarse a menos que fuese absolutamente necesario. Además, las malas noticias no deberían guardarse en secreto, sino entregarse al equipo tan pronto como fuese posible (mientras haya tiempo para reaccionar de un modo racional y controlado).



los equipos formados son lo ideal, pero no es fácil conseguirlos. Como mínimo, esté seguro de evitar un «entorno tóxico».

Aunque la frustración tiene muchas causas, los desarrolladores de software a menudo la sienten cuando pierden la autoridad para controlar la situación. Un equipo de software puede evitar la frustración si recibe tanta responsabilidad para la toma de decisiones como sea posible. Cuanto más control se le de al equipo para tomar decisiones técnicas y del proceso, menos frustración sentiran los miembros del equipo.

Una elección inapropiada del proceso del software (p.ej., tareas innecesarias o pesadas, pobre elección de los productos del trabajo) puede ser evitada de dos formas: (1) estando seguros de que las características del software a construir se ajustan al rigor del proceso elegido, y (2) permitiendo al equipo seleccionar el proceso (con el reconocimiento completo de que, una vez elegido, el equipo tiene la responsabilidad de entregar un producto de alta calidad).

El gestor de proyectos de software, trabajando junto con el equipo, debería refinar claramente los roles y las responsabilidades antes del comienzo del proyecto. El equipo debería establecer su propios mecanismos para la responsabilidad (las revisiones técnicas formales[3] son una forma para realizar esto) y definir una serie de enfoques correctivos cuando un miembro del equipo falla en el desarrollo.

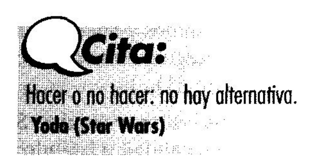
Todo equipo de software experimenta pequeños fallos. La clave para eliminar una atmósfera de fallos será establecer técnicas basadas en el equipo para retroalimentar y solucionar el problema. Además, cualquier fallo de un miembro del equipo debe ser considerado como un fallo del equipo. Esto lleva a un acercamiento del equipo a la acción correctiva, en lugar de culpar y desconfiar, que ocurre con rapidez en equipos tóxicos.

¿Cómo debemos evitar «toxinas» que con frecuencia infectan un equipo de software?

Además de las cinco toxinas descritas por Jackman, un equipo de software a menudo se enfrenta con los rasgos humanos diferentes de sus miembros. Algunos miembros del equipo son extrovertidos, otros son introvertidos. Algunas personas recogen información intuitivamente, separando amplios conceptos de hechos dispares. Otros procesan la información linealmente, reuniendo y organizando detalles minuciosos de los datos proporcionados. Algunos miembros del equipo toman las decisiones apropiadas solamente cuando se presenta un argumento lógico, de un modo ordenado. Otros son intuitivos, pudiendo tomar una decisión basándose en sus «sensaciones». Algunos desarrolladores prefieren una planificación detallada compuesta por tareas organizadas que les permita lograr el cierre para algún elemento de un proyecto. Otros prefieren un entorno más espontáneo donde aspectos abiertos son correctos. Algunos trabajan duro para tener las cosas hechas mucho antes de la fecha de un hito, de ese modo eliminan la presión cuando se aproximan a las fechas, mientras que otros están apurados por las prisas para hacer la entrega en el Último minuto. Un estudio detallado de la psicología de estos rasgos y de las formas en las que un jefe de equipo cualificado puede ayudar a la gente con rasgos opuestos para trabajar juntos está fuera del ámbito de éste libro[4]. Sin embargo, es importante destacar que el reconocimiento de las diferencias humanas es el primer paso hacia la creación de equipos que cuajan.

3.2.4. Aspectos sobre la coordinación y la comunicación

Hay muchos motivos por los que los proyectos de software pueden tener problemas. La *escala* (tamaño) de muchos esfuerzos de desarrollo es grande, conduciendo a complejidades, confusión y dificultades significativas para coordinar a los miembros del equipo. La *incertidumbre* es corriente, dando como resultado un continuo flujo de cambios que impactan al equipo del proyecto. La *interoperatividad* se ha convertido en una característica clave de muchos sistemas. El software nuevo debe comunicarse con el anterior y ajustarse a restricciones predefinidas impuestas por el sistema o el producto.



Estas características del software moderno —escala, incertidumbre e interoperatividad —son aspectos de la vida. Para enfrentarse a ellos eficazmente, un equipo de ingeniería del software debe establecer métodos efectivos para coordinar a la gente que realiza el trabajo. Para lograr esto se deben establecer mecanismos de comunicación formales e informales entre los miembros del equipo y entre múltiples equipos. La comunicación formal se lleva a cabo «por escrito, con reuniones organizadas y otros canales de comunicación relativamente no interactivos e impersonales» [KRA95]. La comunicación informal es más personal. Los miembros de un equipo de software comparten ideas de por sí, piden ayuda a medida que surgen los problemas e interactúan los unos con los otros diariamente.

[3] Las revisiones técnicas formales se tratan con detalle en el Capitulo 8.

[4] Se puede encontrar una excelente introducción a estos temas relacionados con los equipos de proyectos de software en [FER98]

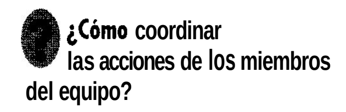


FIGURA 3.1. Valor y empleo de técnicas de coordinación y comunicación.

Kraul y Streeter [KRA95] examinan una colección de técnicas de coordinación de proyectos que se categorizan de la siguiente manera:

Formal, enfoque impersonal. Incluyen documentos de ingeniería del software y entregas (incluyendo el código fuente), memorandos técnicos, hitos del proyecto, planificaciones del programa y herramientas de control del proyecto (Capítulo 7), peticiones de cambios y documentación relativa (Capítulo 9), informes de seguimiento de errores e información almacenada (Capítulo 31).

Formal, procedimientos interpersonales. Se centra en las actividades de garantía de calidad (Capítulo **8)** aplicada a productos de ingeniería del software. Estos incluyen reuniones de revisión de estado ***e*** inspecciones de diseño y de código.



Informal, procedimientos interpersonales. Incluyen reuniones de grupo para la divulgación de información y resolución de problemas así como «definición de requisitos y del personal de desarrollo».

Comunicación electrónica. Comprende correo electrónico, boletines de noticias electrónicos y, por extensión, sistemas de videoconferencia.

Red interpersonal. Discusiones informales con **los** miembros del equipo y con personas que no están en el proyecto pero que pueden tener experiencia ***o*** una profunda visión que puede ayudar a **los** miembros del equipo.

Para valorar la eficacia de estas técnicas para la coordinación de proyectos, Kraul y Streeter estudiaron *65* proyectos de software con cientos de personas implicadas. La Figura **3.1** (adaptada de [KRA95]) expresa el valor y empleo de las técnicas de coordinación apuntadas anteriormente. En la figura, el valor percibido (clasificado en una escala de siete puntos) de varias técnicas de comunicación y coordinación es contrastado con **su** frecuencia de empleo en un proyecto. Las técnicas situadas por encima de la línea de regresión fueron «juzgadas como relativamente valiosas, dado la cantidad de veces que se emplearon» [KRA95]. Las técnicas situadas por debajo de la línea se consideraron de menor valor. Es interesante resaltar que las redes interpersonales fueron catalogadas como las técnicas de mayor valor de coordinación y de comunicación. Es también importante hacer notar que los primeros mecanismos de garantía de calidad del software (requisitos y revisiones de diseño) parecieron tener más valor que evaluaciones posteriores de código fuente (inspecciones de código).

3.3 PRODUCTO

El gestor de un proyecto de software se enfrenta a un dilema al inicio de **un** proyecto de ingeniería del software. Se requieren estimaciones cuantitativas y un plan organizado, pero no se dispone de información sólida. Un análisis detallado de los requisitos del software proporcionaría la información necesaria para las estimaciones, pero el análisis a menudo lleva semanas o meses. **Aún** peor, **los** requisitos pueden ser fluidos, cambiando regularmente a medida que progresa el proyecto. Y, aún así, se necesita un plan «¡ya!».

Por tanto, debemos examinar el producto y el problema a resolver justo al inicio del proyecto. Por lo menos se debe establecer el ámbito del producto y delimitarlo.

3.3.1. Ámbito del software

La primera actividad de gestión de un proyecto de software es determinar ***el** ámbito **del** software.* El ámbito se define respondiendo a las siguientes 'cuestiones:



*Si no puede **delimitar una** característica del software que intenta **construir,** considere **la** característica como un riesgo **principal** del proyecto.*

Contexto. ¿Cómo encaja el software a construir en un sistema, producto o contexto de negocios mayor y qué limitaciones se imponen como resultado del contexto?

Objetivos de información. ¿Qué objetos de datos visibles al cliente (Capítulo 11) se obtienen del software? ¿Qué objetos de datos son requeridos de entrada?

Función y rendimiento. ¿Qué función realiza el software para transformar la información de entrada en una salida? ¿Hay características de rendimiento especiales que abordar?

El ámbito de un proyecto de software debe ser unívoco y entendible a niveles de gestión y técnico. Los enunciados del ámbito del software deben estar delimitados.

Es decir, los datos cuantitativos (por ejemplo: número de usuarios simultáneos, tamaño de la lista de correo, máximo tiempo de respuesta permitido) se establecen explícitamente; se anotan las limitaciones (por ejemplo: el coste del producto limita el tamaño de la memoria) y se describen los factores de reducción de riesgos (por ejemplo: los algoritmos deseados se entienden muy bien si están disponibles en C++).

3.3.2. Descomposición del problema

La descomposición del problema, denominado a veces *particionado* o *elaboración del problema,* es una actividad que se asienta en el núcleo del análisis de requisitos del software (Capítulo 11). Durante la actividad de exposición del ámbito no se intenta descomponer el problema totalmente. **Más** bien, la descomposición se aplica en dos áreas principales: (1) la funcionalidad que debe entregarse y (2) el proceso que se empleará para entregarlo.



Para desarrollar un plan de proyecto razonable, tiene que descomponer funcionalmente el problema a resolver

Los seres humanos tienden a aplicar la estrategia de divide y vencerás cuando se enfrentan a problemas complejos. Dicho de manera sencilla, un problema complejo se parte en problemas más pequeños que resultan más manejables. Ésta es la estrategia que se aplica al inicio de la planificación del proyecto.

Las funciones del software, descritas en la exposición del ámbito, se evalúan y refinan para proporcionar más detalle antes del comienzo de la estimación (Capítulo 5). Puesto que ambos, el coste y las estimaciones de la planificación temporal, están orientados funcionalmente, un pequeño grado de descomposición suele ser útil.

Por ejemplo, considere un proyecto que construirá un nuevo procesador de textos. Entre las características peculiares del producto están: la introducción de información a través de la voz así como del teclado; características extremadamente sofisticadas de «edición automática de copia»; capacidad de diseño de página; indexación automática y tabla de contenido, y otras. El gestor del proyecto debe establecer primero la exposición del ámbito que delimita estas características (así como otras funciones más normales, como edición, administración de archivos, generación de documentos). Por ejemplo, ¿requerirá la introducción de información mediante voz «entrenamiento» por parte del usuario? ¿Qué capacidades específicas proporcionará la característica de editar copias? ¿Exactamente cómo será de sofisticado la capacidad de diseño de página?

Referencia cruzada

En el Capítulo 12 se presento una técnica útil paro descomponer el problema, llamada «análisis gramatical».

A medida que evoluciona la exposición del ámbito, un primer nivel de partición ocurre de forma natural. El equipo del proyecto sabe que el departamento de marketing ha hablado con clientes potenciales y ha averiguado que las siguientes funciones deberían ser parte de la edición automática de copia: (1) comprobación ortográfica; (2) comprobación gramatical; (3) comprobación de referencias para documentos grandes (p. ej.: ¿se puede encontrar una referencia a una entrada bibliográfica en la lista de entradas de la bibliografía?), y (4) validación de referencias de sección y capítulo para documentos grandes. Cada una de estas características representa una subfunción para implernentar en software. Cada una puede ser aún más refinada si la descomposición hace más fácil la planificación.

3.4 PROCESO

Las fases genéricas que caracterizan el proceso de software definición, desarrollo y soporte — son aplicables a todo software. El problema es seleccionar el modelo de proceso apropiado para la ingeniería del software que debe aplicar el equipo del proyecto. En el Capítulo 2 se estudió una gran gama de paradigmas de ingeniería del software:

- el modelo secuencia1 lineal
- el modelo de prototipo
- el modelo DRA
- el modelo incremental
- el modelo en espiral
- el modelo en espiral WINWIN
- el modelo de desarrollo basado (ensamblaje) en componentes
- el modelo de desarrollo concurrente
- el modelo de métodos formales
- el modelo de técnicas de cuarta generación



*Uno vez eleyido el modelo de proceso, ocompóñelo con el mínimo conjunto de **tareas de trabajo** y productos que desembocaronen un producto de **alta calidad** - evite la **capacidad** de destrucción del procese.*

El gestor del proyecto debe decidir qué modelo de proceso es el más adecuado para (1) los clientes que han solicitado el producto y la gente que realizará el trabajo; *(2)* las características del producto en sí, y *(3)* el entorno del proyecto en el que trabaja el equipo de software. Cuando se selecciona un modelo de proceso, el equipo define entonces un plan de proyecto preliminar basado en un conjunto de actividades estructurales. Una vez establecido el plan preliminar, empieza la descomposición del proceso. Es decir, se debe crear un plan completo reflejando las tareas requeridas a las personas para cubrir las actividades estructurales. Exploramos estas actividades brevemente en las secciones que siguen y presentamos una visión **más** detallada en el Capítulo 7.

3.4.1. Maduración del producto y del proceso

La planificación de un proyecto empieza con la maduración del producto y del proceso. Todas las funciones que se deben tratar dentro de un proceso de ingeniería por el equipo de software deben pasar por el conjunto de actividades estructurales que se han definido para una organización de software. Asuma que la organización ha adoptado el siguiente conjunto de actividades estructurales (Capítulo 2):

- *Comunicación con el cliente* — tareas requeridas para establecer la obtención de requisitos eficiente entre el desarrollador y el cliente.
- *Planificación* — tareas requeridas para definir los recursos, la planificación temporal del proyecto y cualquier información relativa a él.



Recuerde.... las actividades estructurales se aplican en todos los proyectos- no hoy excepciones.

- *Análisis del riesgo* — tareas requeridas para valorar los riesgos técnicos y de gestión.
- *Ingeniería* — tareas requeridas para construir una o más representaciones de la aplicación.
- *Construcción y entrega* — tareas requeridas para construir, probar, instalar y proporcionar asistencia al usuario (por ejemplo: documentación y formación).
- *Evaluación del cliente* — tareas requeridas para obtener información de la opinión del cliente basadas en la evaluación de las representaciones de software creadas durante la fase de ingeniería e implementadas durante la fase de instalación.

Los miembros del equipo que trabajan en cada función aplicarán todas las actividades estructurales. En esencia, se crea una matriz similar a la que se muestra en la Figura 3.2. Cada función principal del problema (la figura contiene las funciones para el software procesador de textos comentado anteriormente) se lista en la columna de la izquierda. Las actividades estructurales se listan en la fila

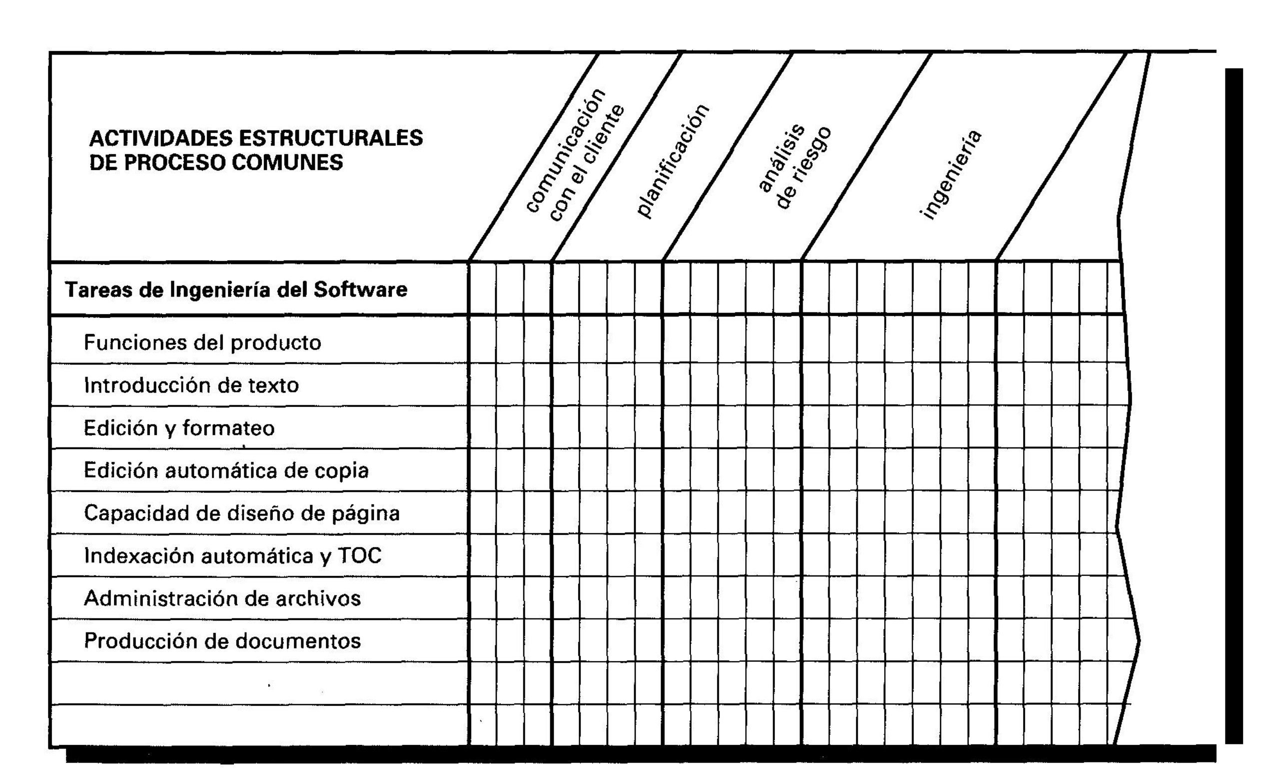


FIGURA 3.2. Maduración del problema y del proceso.

de arriba. Las tareas de trabajo de ingeniería del software (para cada actividad estructural) se introducirían en la fila siguiente[5]. El trabajo del gestor del proyecto (y de otros miembros del equipo) es estimar los requisitos de recursos para cada celda de la matriz, poner fechas de inicio y finalización para las tareas asociadas con cada celda y los productos a fabricar como consecuencia de cada celda. Estas actividades se consideran en los Capítulos 5 y 7.

CLAVE

La descomposición del producto y del proceso se produce simultóneamente con la evolución del plan de proyecto.

3.4.2. Descomposición del proceso

Un equipo de software debería tener un grado significativo de flexibilidad en la elección del paradigma de ingeniería del software que resulte mejor para el proyecto y de las tareas de ingeniería del software que conforman el modelo de proceso una vez elegido. Un proyecto relativamente pequeño similar a otros que se hayan hecho anteriormente se debería realizar con el enfoque secuencia1 lineal. Si hay límites de tiempo muy severos y el problema se puede compartimentar mucho, el modelo apropiado es el DRA (en inglés RAD). Si la fecha límite está tan próxima que no va a ser posible entregar toda la funcionalidad, una estrategia incremental puede ser lo mejor. Similarmente, proyectos con otras características (p. ej.: requisitos inciertos, nuevas tecnologías, clientes difíciles, potencialidad de reutilización) llevarán a la selección de otros modelos de proceso[6].



Aplique siempre la ECP (Estructura Común de Proceso), sin tener en cuenta el tamaño, criticidad o tipo del proyecto. Las tareas pueden variar pero la ECP no.

Una vez que se ha elegido el modelo de proceso, la estructura común de proceso (ECP) se adapta a él. En todos los casos, el ECP estudiado anteriormente en este capítulo –comunicación con el cliente, planificación, análisis de riesgo, ingeniería, construcción, entrega y evaluación del cliente— puede adaptarse al paradigma. Funcionará para modelos lineales, para modelos iterativos e incrementales, para modelos de evolución e incluso para modelos concurrentes o de ensamblaje de componentes. El ECP es invariable y sirve como base para todo el trabajo de software realizado por una organización.

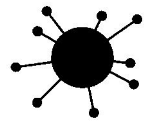
Pero las tareas de trabajo reales sí varían. La descomposición del proceso comienza cuando el gestor del proyecto pregunta: «¿Cómo vamos a realizar esta actividad ECP?». Por ejemplo, un pequeño proyecto, relativamente simple, requiere las siguientes tareas para la actividad de *comunicación con el cliente:*

1. Desarrollar una lista de aspectos que se han de clarificar.
2. Reunirse con el cliente para resolver los aspectos que se han de clarificar.
3. Desarrollar conjuntamente una exposición del ámbito del proyecto.
4. Revisar el alcance del proyecto con todos los implicados.
5. Modificar el alcance del proyecto cuando se requiera.

Estos acontecimientos pueden ocurrir en un periodo de menos de 48 horas. Representan una descomposición del problema apropiado para proyectos pequeños relativamente sencillos.

Ahora, consideremos un proyecto más complejo que tenga un ámbito más amplio y un mayor impacto comercial. Un proyecto como ése podría requerir las siguientes tareas para la actividad de comunicación con el cliente:

1. Revisar la petición del cliente.
2. Planificar y programar una reunión formal con el cliente.
3. Realizar una investigación para definir soluciones propuestas y enfoques existentes.
4. Preparar un «documento de trabajo» y una agenda para la reunión formal.
5. Realizar la reunión.
6. Desarrollar conjuntamente mini-especificaciones que reflejen la información, función y características de comportamiento del software.



Modelo de proceso adaptable.

7. Revisar todas las mini-especificaciones para comprobar su corrección, su consistencia, la ausencia de ambigüedades.
8. Ensamblar las mini-especificaciones en un documento de alcance del proyecto.
9. Revisar ese documento general con todo lo que pueda afectar.
10. Modificar el documento de alcance del proyecto cuando se requiera.

Ambos proyectos realizan la actividad estructural que denominamos *comunicación con el cliente,* pero el equipo del primer proyecto lleva a cabo la mitad de tareas de ingeniería del software que el segundo.

[5] Se hace notar que las tareas se deben adaptar a las necesidades específicas de un proyecto. Las actividades estructurales siempre permanecen igual, pero las tareas se seleccionarán basándose en unos criterios de adaptación. Este tema es discutido más adelante en el Capitulo 7 y en la página Web SEPA 5/e.

[6] Recuerde que las características del proyecto tienen también una fuerte influencia en la estructura del equipo que va a realizar el trabajo. Vea la Sección 3.2.3.

3.5 PROYECTO

Para gestionar un proyecto de software con éxito, debemos comprender qué puede ir mal (para evitar esos problemas) y cómo hacerlo bien. En un excelente documento sobre proyectos de software, John Reel [REE99] define diez señales que indican que un proyecto de sistemas de información está en peligro:

Cita:

Al menos 7 de las 10 señales que indican el fallo de un proyecto de sistemas de información se determinan antes del desarrollo de un diseño o antes de escribir una línea de código.

John Reel

1. La gente del software no comprende las necesidades de los clientes.
2. El ámbito del producto está definido pobremente.
3. Los cambios están mal realizados.
4. La tecnología elegida cambia.
5. Las necesidades del negocio cambian [o están mal definidas].
6. Las fechas de entrega no son realistas.
7. Los usuarios se resisten.
8. Se pierden los patrocinadores [o nunca se obtuvieron adecuadamente].
9. El equipo del proyecto carece del personal con las habilidades apropiadas.
10. Los gestores [y los desarrolladores] evitan buenas prácticas y sabias lecciones.

Los profesionales veteranos de la industria hacen referencia frecuentemente a la regla 90-90 cuando estudian proyectos de software particularmente difíciles: el primer 90 por 100 de un sistema absorbe el 90 por 100 del tiempo y del esfuerzo asignado. El último 10 por 100 se lleva el otro 90 por 100 del esfuerzo y del tiempo asignado [ZAH94]. Los factores que conducen a la regla 90-90 están contenidos en las diez señales destacadas en la lista anterior.

¡Suficiente negatividad! ¿Cómo actúa un gestor para evitar los problemas señalados anteriormente? Reel [REE99] sugiere una aproximación de sentido común a los proyectos de software dividida en cinco partes:

Empezar con el pie derecho. Esto se realiza trabajando duro (muy duro) para comprender el problema a solucionar y estableciendo entonces objetivos y expectativas realistas para cualquiera que vaya a estar involucrado en el proyecto. Se refuerza construyendo el equipo adecuado (Sección 3.2.3) y dando al equipo la autonomía, autoridad y tecnología necesaria para realizar el trabajo.

Mantenerse. Muchos proyectos no realizan un buen comienzo y entonces se desintegran lentamente. Para mantenerse, el gestor del proyecto debe proporcionar incentivos para conseguir una rotación del personal mínima, el equipo debería destacar la calidad en todas las tareas que desarrolle y los gestores veteranos deberían hacer todo lo posible por permanecer fuera de la forma de trabajo del equipo[7].

Seguimiento del Progreso. Para un proyecto de software, el progreso se sigue mientras se realizan los productos del trabajo (por ejemplo, especificaciones, código fuente, conjuntos de casos de prueba) y se aprueban (utilizando revisiones técnicas formales) como parte de una actividad de garantía de calidad. Además, el proceso del software y las medidas del proyecto (capítulo 4) pueden ser reunidas y utilizadas para evaluar el progreso frente a promedios desarrollados por la organización de desarrollo de software.

Tomar Decisiones Inteligentes. En esencia, las decisiones del gestor del proyecto y del equipo de software deberían «seguir siendo sencillas». Siempre que sea posible, utilice software del mismo comercial o componentes de software existentes; evite personalizar interfaces cuando estén disponibles aproximaciones estándar; identifique y elimine entonces riesgos obvios; asigne más tiempo del que pensaba necesitar para tareas arriesgadas o complejas (necesitará cada minuto).



Referencia Web

Se puede encontrar un gran conjunto de recursos que pueden ayudar tanto a gestores de proyecto experimentodos como a principiantes en: **www.pmi.org, www.4pm.com y www.projectmanagement.com**

Realizar un Análisis «Postmortem» (después de finalizar el proyecto}. Establecer un mecanismo consistente para extraer sabias lecciones de cada proyecto. Evaluar la planificación real y la prevista, reunir y analizar métricas del proyecto de software y realimentar con datos de los miembros del equipo y de los clientes, y guardar los datos obtenidos en formato escrito.

[7] Esta frase implica la reducción al mínimo de la burocracia, y la eliminación tanto de reuniones extrañas como de la adherencia dogmática a las reglas del proceso y del proyecto. El equipo debena estar capacitado para realizar esto.

3.6 EL PRINCIPIO W^{5}HH

En un excelente documento sobre proyectos y proceso del software, Barry Boehm [BOE96] afirma: «... se necesita un principio de organización que haga una simplificación con el fin de proporcionar planes [de proyectos] sencillos para proyectos pequeños». Boehm sugiere un enfoque que trate los objetivos, hitos y planificación, responsabilidades, enfoque técnico y de gestión, y los recursos requeridos del proyecto. Bohem lo llama el principio «WWWWWHH», después de una serie de preguntas (7 cuestiones) que conducen a la definición de las características clave del proyecto y el plan del proyecto resultante:

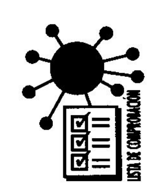
¿Por qué se desarrolla el sistema? La respuesta a esta pregunta permite a todas las partes evaluar la validez de las razones del negocio para el trabajo del software. Dicho de otra forma, ¿justifica el propósito del negocio el gasto en personal, tiempo, y dinero?

¿Qué se realizará y cuándo? La respuesta a estas preguntas ayuda al equipo a establecer la planificación del proyecto identificando las tareas clave del proyecto y los hitos requeridos por el cliente.

¿Qué preguntas necesitan ser respondidas para desarrollar un Plan de Proyecto?

¿Quién es el responsable de una función? Antes en este capítulo, señalamos que el papel y la responsabilidad de cada miembro del equipo de software debe estar definida. La respuesta a la pregunta ayuda a cumplir esto.

¿Dónde están situados organizacionalmente? No todos los roles y responsabilidades residen en el equipo de software. El cliente, los usuarios, y otros directivos también tienen responsabilidades.



Plan de Proyecto de Software

¿Cómo estará realizado el trabajo desde el punto de vista técnico y de gestión? Una vez establecido el ámbito del producto, se debe definir una estrategia técnica y de gestión para el proyecto.

¿Qué cantidad de cada recurso se necesita? La respuesta a esta pregunta se deriva de las estimaciones realizadas (Capítulo 5) basadas en respuestas a las preguntas anteriores.

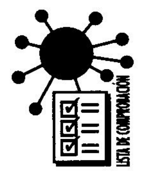
El principio W^{5}HH de Boehm es aplicable sin tener en cuenta el tamaño o la complejidad del proyecto de software. Las preguntas señaladas proporcionan un perfil de planificación al gestor del proyecto y al equipo de software.

3.7 PRÁCTICAS CRÍTICAS

El Concilio[8] Airlie ha desarrollado una lista de «prácticas críticas de software para la gestión basada en el rendimiento». Estas prácticas son «utilizadas de un modo consistente por, y consideradas críticas por, organizaciones y proyectos de software de mucho éxito cuyo rendimiento "final" es más consistente que los promedios de la industria» [AIR99]. En un esfuerzo por permitir a una organización de software determinar si un proyecto específico ha implementado prácticas críticas, el Concilio Airlie ha desarrollado un conjunto de preguntas de «Visión Rápida» [AIR99] para un proyecto[9]:

Gestión formal del riesgo. ¿Cuáles son los diez riesgos principales para este proyecto? Para cada uno de los riesgos ¿cuál es la oportunidad de que el riesgo se convierta en un problema y cuál es el impacto si lo hace?

Coste empírico y estimación de la planificación. ¿Cuál es el tamaño actual estimado de la aplicación de software (sin incluir el software del sistema) que será entregada en la operación? ¿Cómo se obtuvo?



Visión rápida del Proyecto Airlie

[8] El Concilio Airlie es un equipo de expertos en ingeniería del software promocionado por el Departamento de Defensa U.S. ayudando en el desarrollo de directrices para prácticas importantes en la gestión de proyectos de software y en la ingeniería del Software.

[9] Solo se destacan aquí aquellas practicas críticas relacionadas con la «integridad del proyecto)).Otras practicas mejores se trataran en capítulos posteriores.

Gestión de proyectos basada en métricas. ¿Dispone de un programa de métricas para dar una primera indicación de los problemas del desarrollo? Si es así, ¿cuál es la volatilidad de los requisitos actualmente?

Seguimiento del valor ganado. ¿Informa mensualmente de las métricas del valor ganado...? Si es así, ¿están calculadas estas métricas desde una red de actividades de tareas para el esfuerzo total a la próxima entrega?

Seguimiento de defectos frente a objetivos de calidad. ¿Realiza el seguimiento e informa periódicamente del número de defectos encontrados en cada prueba de inspección [revisión técnica formal] y ejecución desde el principio del programa y del número de defectos que se corrigen y se producen en la actualidad?

Gestión del programa del personal. ¿Cuál es la media de rotación de la plantilla en los tres Últimos meses por cada uno de los distribuidores/desarrolladores involucrados en el desarrollo del software para este sistema?

Si un equipo de proyectos de software no puede responder a estas preguntas, o las responde inadecuadamente, se debe realizar una revisión completa de las prácticas del proyecto. Cada una de las prácticas críticas señaladas anteriormente se tratan con detalle a lo largo de la Parte II de este libro.

RESUMEN

La gestión de proyectos de software es una actividad protectora dentro de la ingeniería del software. Empieza antes de iniciar cualquier actividad técnica y continúa a lo largo de la definición, del desarrollo y del mantenimiento del software.

Hay cuatro « P's » que tienen una influencia sustancial en la gestión de proyectos de software —personal, producto, proceso y proyecto—. El personal debe organizarse en equipos eficaces, motivados para hacer un software de alta calidad y coordinados para alcanzar una comunicación efectiva. Los requisitos del producto deben comunicarse desde el cliente al desarrollador, dividirse (descomponerse) en las partes que lo constituyen y distribuirse para que trabaje el equipo de software. El proceso debe adaptarse al personal y al problema. Se selecciona una estructura común de proceso, se aplica un paradigma apropiado de ingeniería del software y se elige un conjunto de tareas para completar el trabajo. Finalmente, el proyecto debe organizarse de una manera que permita al equipo de software tener éxito.

El elemento fundamental en todos los proyectos de software es el personal. Los ingenieros del software pueden organizarse en diferentes organigramas de equipo que van desde las jerarquías de control tradicionales a los equipos de «paradigma abierto». Se pueden aplicar varias técnicas de coordinación y comunicación para apoyar el trabajo del equipo. En general, las revisiones formales y las comunicaciones informales persona a persona son las más valiosas para los profesionales.

La actividad de gestión del proyecto comprende medición y métricas, estimación, análisis de riesgos, planificación del programa, seguimiento y control. Cada uno de estos aspectos se trata en los siguientes capítulos.

REFERENCIAS

[AIR99] Airlie Council, «Performanced Based Management: The Program Manager's Guide Based on the 16-Point Plan and Related Metrics», Draft Report, 8 de Marzo, 1999.

[BAK72] Baker, F.T., «Chief Programmer Team Management of Production Programming», *IBM Systems Journal,* vol. 11, n.º 1, 1972, pp. 56-73.

[BOE96] Boehm, B., «Anchoring the Software Process», *IEEE Software,* vol. 13, n.º 4, Julio de 1996, pp. 73-82.

[CON93] Constantine, L., «Work Organization: Paradigms for Project Management and Organization», *CACM,* vol. 36, n.º 10, Octubre de 1993, pp. 34-43.

[COU80] Cougar, J., y R. Zawacky, *Managing and Motivating Computer Personnel,* Wiley, 1980.

[CUR88] Curtis, B., *et al,* «A Field Study of the Software Design Process for Large Systems», *IEEE Trans. Software Engineering,* vol. 31, n.º 11, Noviembre de 1988, pp. 1268-1287.

[CUR94] Curtis, B., *et al., People Management Capability Maturity Model,* Software Engineering Institute, Pittsburgh, PA, 1994.

[DEM98] DeMarco, T., y T. Lister, *Peopleware,* 2.ª edición, Dorset House, 1998.

[EDG95] Edgemon, J., «Right Stuff How to Recognize It When Selecting a Project Manager», *Application Development Trends,* vol. 2, n.º 5, Mayo de 1995, pp. 37-42.

[FER98] Ferdinandi, P. L., «Facilating Communication», *IEEE Software,* Septiembre de 1998, pp. 92-96.

[JAC98] Jackman, M., «Homeopathic Remedies for Team Toxicity», *IEEE Software,* Julio de 1998, pp. 43-45.

[KRA95] Kraul, R., y L. Streeter, «Coordination in Software Development», *CACM*, vol. 38, n.º 3, Marzo de 1995, pp. 69-81.

[MAN81] Mantei, M., «The Effect of Programming Team Structures on Programming Tasks», *CACM*, vol. 24, n.º 3, Marzo de 1981, pp. 106-113.

[PAG85] Page-Jones, M., *Practical Project Management*, Dorset House, 1985, p. VII.

[REE99] Reel, J.S., «Critical Success Factors in Software Projects», *IEEE Software*, Mayo de 1999, pp. 18-23.

[WEI86] Weinberg, G., *On Becoming a Technical Leader*, Dorset House, 1986.

[WIT94] Whitaker, K., *Managing Software Maniacs*, Wiley, 1994.

[ZAH94] Zahniser, R., «Timeboxing for Top Team Performance», *Software Development*, Marzo de 1994, pp. 35-38.

PROBLEMAS Y PUNTOS A CONSIDERAR

3.1. Basándose en la información contenida en este capítulo y en su propia experiencia, desarrolle «diez mandamientos» para potenciar a los ingenieros del software. Es decir, haga una lista con las diez líneas maestras que lleven al personal que construye software a su máximo potencial.

3.2. El Modelo de Madurez de Capacidad de Gestión de Personal (MMCGP) del Instituto de Ingeniería del Software hace un estudio organizado de las «áreas clave prácticas (ACP)» que cultivan los buenos profesionales del software. Su profesor le asignará una ACP para analizar y resumir.

3.3. Describa tres situaciones de la vida real en las que el cliente y el usuario final son el mismo. Describa tres situaciones en que son diferentes.

3.4. Las decisiones tomadas por una gestión experimentada pueden tener un impacto significativo en la eficacia de un equipo de ingeniería del software. Proporcione cinco ejemplos para ilustrar que es cierto.

3.5. Repase el libro de Weinberg [WEI86] y escriba un resumen de dos o tres páginas de los aspectos que deberían tenerse en cuenta al aplicar el modelo MOI.

3.6. Se le ha nombrado gestor de proyecto dentro de una organización de sistemas de información. Su trabajo es construir una aplicación que es bastante similar a otras que ha construido su equipo, aunque ésta es mayor y más compleja. Los requisitos han sido detalladamente documentados por el cliente. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo(s) de proceso de software elegiría y por qué?

3.7. Se le ha nombrado gestor de proyecto de una pequeña compañía de productos software. Su trabajo consiste en construir un producto innovador que combine hardware de realidad virtual con software innovador. Puesto que la competencia por el mercado de entretenimiento casero es intensa, hay cierta presión para terminar el trabajo rápidamente. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo(s) de proceso de software elegiría y por qué?

3.8. Se le ha nombrado gestor de proyecto de una gran compañía de productos software. Su trabajo consiste en dirigir la versión de la siguiente generación de su famoso procesador de textos. Como la competencia es intensa, se han establecido y anunciado fechas límite rígidas. ¿Qué estructura de equipo elegiría y por **qué?** ¿Qué modelo(s) de proceso de software elegiría y por **qué?**

3.9. Se le ha nombrado gestor de proyecto de software de una compañía que trabaja en el mundo de la ingeniería genética. Su trabajo es dirigir el desarrollo de un nuevo producto de software que acelere el ritmo de impresión de genes. El trabajo es orientado a I+D, pero la meta es fabricar el producto dentro del siguiente año. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo(s) de proceso de software elegiría y por **qué?**

3.10. Como muestra la Figura 3.1, basándose en los resultados de dicho estudio, los documentos parecen tener más uso que valor. ¿Por **qué** cree que pasó esto y qué se puede hacer para mover el punto documentos por encima de la línea de regresión en el gráfico? Es decir, ¿qué se puede hacer para mejorar el valor percibido de los documentos?

3.11. Se le ha pedido que desarrolle una pequeña aplicación que analice todos los cursos ofrecidos por la universidad e informe de las notas medias obtenidas en los cursos (para un periodo determinado). Escriba una exposición del alcance que abarca este problema.

3.12. Haga una descomposición funcional de primer nivel de la función diseño de página tratado brevemente en la Sección 3.3.2.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Una excelente serie de cuatro volúmenes escrito por Weinberg *(Quality Software Management,* Dorset House, 1992, 1993, 1994, 1996) introduce los conceptos básicos sobre sistemas y conceptos de gestión, explica cómo usar mediciones eficazmente y menciona la «acción congruente», la habilidad de «encajar» las necesidades del gestor, del personal técnico y las del negocio. Proporciona información útil tanto a los gestores noveles como a los experimentados. **Brooks** *(The Mythical Man-Month,* Aniversary Edition, Addison-Wesley, 1995) ha actualizado su clásico libro para proporcionar una nueva visión profunda de los aspectos del proyecto de software y de su gestión. Purba y Shah *(How to Manage a Successful Software Project,* Wiley, 1995) presentan un número de estudios de casos de proyectos que indican por qué unos fracasaron y otros fueron un éxito. Bennatan *(Software Project Management in a Client/Server Environment,* Wiley, 1995) estudia aspectos específicos de gestión asociados con el desarrollo de sistemas cliente/servidor.

Se puede argumentar que el aspecto más importante de la gestión del proyecto de software es la gestión de personal. El

libro definitivo sobre este tema lo escribieron DeMarco y Lister [DEM98], pero se han publicado en los Últimos años los siguientes libros donde se examina **su** importancia:

Beaudouin-Lafon, M., *Computer Supported Cooperative Work,* Wiley-Liss, 1999.

Carmel, E., *Global Software Teams: Collaborating Across Borders and Time Zones,* Prentice Hall, 1999.

Humphrey, W. S., *Managing Technical People: Innovation, Teamwork, and the Software Process,* Addison-Wesley 1997.

Humphrey, W. S., *Introduction to the Team of Software Process,* Addison-Wesley, 1999.

Jones, P. H., *Handbook of Team Design: A Practitioner's Guide to Team Systems Development,* McGraw-Hill, 1997.

Karolak, D. S., *Global Software Development: Managing Virtual Teams and Environments,* IEEE Computer Society, 1998.

Mayer, M., *The Virtual Edge: Embracing Technology for Distributed Project Team Success,* Project Management Institute Publications, 1999.

Otro excelente libro de Weinberg [WEI86] es lectura obligada para todo gestor de proyecto y jefe de equipo. Le dará una visión interna y directrices para realizar **su** trabajo más eficazmente. House *(The Human Side of Project Management,* Addison-Wesley, 1988) y Crossby *(Running Things: The art of Making Things Happen,* McGraw-Hill, 1989) proporcionan directrices prácticas para gestores que deban tratar con problemas humanos **y** técnicos.

Aunque no están relacionados específicamente con **el** mundo del software, y algunas veces sufren demasiadas simplificaciones y amplias generalizaciones, los libros de gran éxito de Drucker *(Management Challenges for the 21st Century,* Harperbusines, 1999), Buckingham y Coffman *(First, Break All the Rules: What the World's Greatest Managers Do Differently,* Simon & Schuster, 1999) y Christensen *(The Innovator's Dilemma,* Harvard Business School Press, 1997) enfatizan «nuevas reglas» definidas por una economía que cambia con rapidez. Viejos títulos como *The One Minute Manager* e *In Search of Excellence* continúan proporcionando enfoques valiosos que pueden ayudarle **a** gestionar los temas relacionados con el personal de un modo más eficiente.

En Internet están disponibles una gran variedad de fuentes de información relacionadas con temas de gestión de proyectos de software. Se puede encontrar una lista actualizada con referencias a sitios (páginas) web que son relevantes para los proyectos de software en **http://www.pressman5.com.**

CAPÍTULO

4 PROCESO DE SOFTWARE Y MÉTRICAS DE PROYECTOS

LA medición es fundamental para cualquier disciplina de ingeniería, y la ingeniería del software no es una excepción. La medición nos permite tener una visión más profunda proporcionando un mecanismo para la evaluación objetiva. Lord Kelvin en una ocasión dijo:

Cuando pueda medir lo que está diciendo y expresarlo con números, ya conoce algo sobre ello; cuando no pueda medir, cuando no pueda expresar lo que dice con números, su conocimiento es precario y deficiente: puede ser el comienzo del conocimiento, pero en sus pensamientos, apenas está avanzando hacia el escenario de la ciencia.

La comunidad de la ingeniería del software ha comenzado finalmente a tomarse en serio las palabras de Lord Kelvin. Pero no sin frustraciones y sí con gran controversia.

Las *métricas del software* se refieren a un amplio elenco de mediciones para el software de computadora. La medición se puede aplicar al proceso del software con el intento de mejorarlo sobre una base continua. Se puede utilizar en el proyecto del software para ayudar en la estimación, el control de calidad, la evaluación de productividad y el control de proyectos. Finalmente, el ingeniero de software puede utilizar la medición para ayudar a evaluar la calidad de los resultados de trabajos técnicos y para ayudar en la toma de decisiones táctica a medida que el proyecto evoluciona.

VISTAZO RÁPIDO

¿Qué es? El proceso del software y las métricas del producto son una medida cuantitativa que permite a la gente del software tener una visión profunda de la eficacia del proceso del software y de los proyectos que dirigen utilizando el proceso como un marco de trabajo. Se reúnen los datos básicos de calidad y productividad. Estos datos son entonces analizados, comparados con promedios anteriores, y evaluados para determinar las mejoras en la calidad y productividad. Las métricas son también utilizadas para señalar áreas con problemas de manera que se puedan desarrollar los remedios y mejorar el proceso del software.

¿Quién lo hace? Las métricas del software son analizadas y evaluadas por los administradores del software. A menudo las medidas son reunidas por los ingenieros del software.

¿Por qué es importante? Si no mides, sólo podrás juzgar basándote en una evaluación subjetiva. Mediante la medición, se pueden señalar las tendencias (buenas o malas), realizar mejores estimaciones, llevar a cabo una verdadera mejora sobre el tiempo.

¿Cuáles son los pasos? Comenzar definiendo un conjunto limitado de medidas de procesos, proyectos y productos que sean fáciles de recoger. Estas medidas son a menudo normalizadas utilizando métricas orientadas al tamaño o a la función. El resultado se analiza y se compara con promedios anteriores de proyectos similares realizados con la organización. Se evalúan las tendencias y se generan las conclusiones.

¿Cuál es el producto obtenido? Es un conjunto de métricas del software que proporcionan una visión profunda del proceso y de la comprensión del proyecto.

¿Cómo puedo estar *seguro* de que lo he hecho correctamente? Aplicando un plan de medición sencillo pero consistente, que nunca utilizaremos para evaluar, premiar o castigar el rendimiento individual.

Dentro del contexto de la gestión de proyectos de software, en primer lugar existe una gran preocupación por las métricas de productividad y de calidad —medidas «de salida» (finalización) del desarrollo del software, basadas en el esfuerzo y tiempo empleados, y medidas de la «utilidad» del producto obtenido—.

Park, Goethert y Florac [PAR96] tratan en su guía de la medición del software las razones por las que medimos:

Hay cuatro razones para medir los procesos del software, los productos y los recursos:

- caracterizar
- evaluar
- predecir
- mejorar

Caracterizamos para comprender mejor los procesos, los productos, los recursos y los entornos y para establecer las líneas base para las comparaciones con evaluaciones futuras.

Evaluamos para determinar el estado con respecto al diseño. Las medidas utilizadas son los sensores que nos permiten conocer cuándo nuestros proyectos y nuestros procesos están perdiendo la pista, de modo que podamos ponerlos bajo control. También evaluamos para valorar la consecución de los objetivos de calidad y para evaluar el impacto de la tecnología y las mejoras del proceso en los productos y procesos.

Predecimos para poder planificar. Realizar mediciones para la predicción implica aumentar la comprensión de las relaciones entre los procesos y los productos y la construcción de modelos de estas relaciones, por lo que los valores que observamos para algunos atributos pueden ser utilizados para predecir otros. Hacemos esto porque queremos establecer objetivos alcanzables para el coste, planificación, y calidad - d emanera que se puedan aplicar los recursos apropiados—. Las medidas de predicción también son la base para la extrapolación de tendencias, con lo que la estimación para el coste, tiempo y calidad se puede actualizar basándose en la evidencia actual. Las proyecciones y las estimaciones basadas en datos históricos también nos ayudan a analizar riesgos y a realizar intercambios diseño/coste.

Medimos para mejorar cuando recogemos la información cuantitativa que nos ayuda a identificar obstáculos, problemas de raíz, ineficiencias y otras oportunidades para mejorar la calidad del producto y el rendimiento del proceso.

4.1 MEDIDAS, MÉTRICAS E INDICADORES

Aunque los términos *medida, medición y métricas* se utilizan a menudo indistintamente, es importante destacar las diferencias sutiles entre ellos. Como los términos «medida» y «medición» se pueden utilizar como un nombre o como un verbo, las definiciones de estos términos se pueden confundir. Dentro del contexto de la ingeniería del software, una *medida* proporciona una indicación cuantitativa de la extensión, cantidad, dimensiones, capacidad o tamaño de algunos atributos de un proceso o producto. La *medición* es el acto de determinar una medida. El *IEEE Standard Glossary of Software Engineering Terms* [IEEE93] define *métrica* como «una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado».

Cuando, simplemente, se ha recopilado un solo aspecto de los datos (por ejemplo: el número de errores descubiertos en la revisión de un módulo), se ha establecido una medida. La medición aparece como resultado de la recopilación de uno o varios aspectos de los datos (por ejemplo: se investiga un número de revisiones de módulos para recopilar medidas del número de errores encontrados durante cada revisión). Una métrica del software relata de alguna forma las medidas individuales sobre algún aspecto (por ejemplo: el número medio de errores encontrados por revisión o el número medio de errores encontrados por persona y hora en revisiones[1]).

Un ingeniero del software recopila medidas y desarrolla métricas para obtener indicadores. Un *indicador* es una métrica o una combinación de métricas que proporcionan una visión profunda del proceso del software, del proyecto de software o del producto en sí [RAG95]. Un indicador proporciona una visión profunda que permite al gestor de proyectos o a los ingenieros de software ajustar el producto, el proyecto o el proceso para que las cosas salgan mejor.



No todo lo que se puede contar cuenta, y no todo lo que cuenta se puede contar.
Albert Einstein

Por ejemplo, cuatro equipos de software están trabajando en un proyecto grande de software. Cada equipo debe conducir revisiones del diseño, pero puede seleccionar el tipo de revisión que realice. Sobre el examen de la métrica, *errores encontrados por persona-hora consumidas*, el gestor del proyecto notifica que dos equipos que utilizan métodos de revisión más formales presentan un 40 por 100 más de *errores encontrados por persona-hora consumidas* que otros equipos. Suponiendo que todos los parámetros son iguales, esto proporciona al gestor del proyecto un indicador en el que los métodos de revisión más formales pueden proporcionar un ahorro mayor en inversión de tiempo que otras revisiones con un enfoque menos formal. Esto puede sugerir que todos los equipos utilicen el enfoque más formal. La métrica proporciona al gestor una visión más profunda. **Y** además le llevará a una toma de decisiones más fundamentada.

[1] Esto asume que se recopila otra medida, persona y horas gastadas en cada revisión.

4.2 MÉTRICAS EN EL PROCESO Y DOMINIOS DEL PROYECTO

La medición es algo común en el mundo de la ingeniería. Se mide el consumo de energía, el peso, las dimensiones físicas, la temperatura, el voltaje, la relación señal-ruido... la lista es casi interminable. Por desgracia, la medición es mucho menos común en el mundo de la ingeniería del software. Existen problemas para ponerse de acuerdo sobre qué medir y las medidas de evaluación de problemas recopilados.



Se puede descargar una guío completo de métricos del software desde: **www.inv.nasa.gov/SWG/resources/NASA-GB-001-94.pdf**

Se deberían recopilar métricas para que los indicadores del proceso y del producto puedan ser ciertos. Los *indicadores de proceso* permiten a una organización de ingeniería del software tener una visión profunda de la eficacia de un proceso ya existente (por ejemplo: el paradigma, las tareas de ingeniería del software, productos de trabajo e hitos). También permiten que los gestores evalúen lo que funciona y lo que no. Las métricas del proceso se recopilan de todos los proyectos y durante un largo período de tiempo. Su intento es proporcionar indicadores que lleven a mejoras de los procesos de software a largo plazo.

Los *indicadores de proyecto* permiten al gestor de proyectos del software (1) evaluar el estado del proyecto en curso; (2) seguir la pista de los riesgos potenciales: (3) detectar las áreas de problemas antes de que se conviertan en «críticas»; (4) ajustar el flujo y las tareas del trabajo, y *(5)* evaluar la habilidad del equipo del proyecto en controlar la calidad de los productos de trabajo del software.

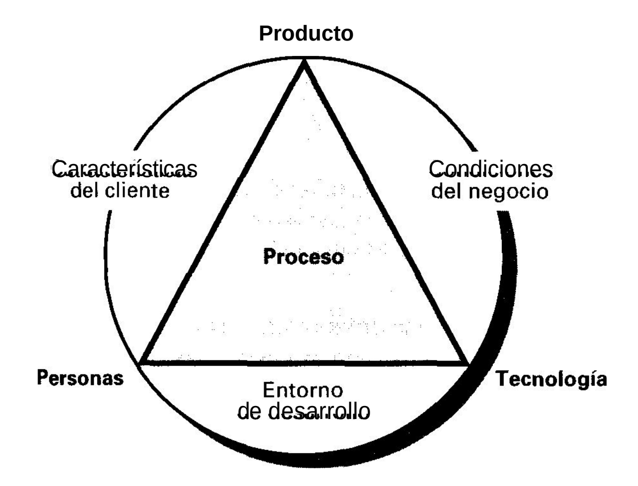


FIGURA 4.1. Determinantes de la calidad del software y de la efectividad de organización (adaptado según [PAU94]).

En algunos casos, se pueden utilizar las mismas métricas del software para determinar tanto el proyecto como los indicadores del proceso. En realidad, las medidas que recopila un equipo de proyecto y las convierte en métricas para utilizarse durante un proyecto también pueden transmitirse a los que tienen la responsabilidad de mejorar el proceso del software. Por esta razón, se utilizan muchas de las mismas métricas tanto en el dominio del proceso como en el del proyecto.

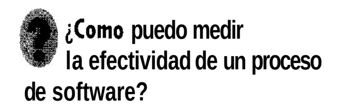
4.2.1. Métricas del proceso y mejoras en el proceso del software

La Única forma racional de mejorar cualquier proceso es medir atributos del proceso, desarrollar un juego de métricas significativas según estos atributos y entonces utilizar las métricas para proporcionar indicadores que conducirán a una estrategia de mejora. Pero antes de estudiar las métricas del software y su impacto en la mejoras de los procesos del software es importante destacar que el proceso es el Único factor de «los controlables al mejorar la calidad del software y su rendimiento como organización» [PAU94].

CLAVE

La habilidad y la motivación del personal realizando el trabajo son los factores más importantes que influyen en la calidad del software.

En la Figura 4.1, el proceso se sitúa en el centro de un triángulo que conecta tres factores con una profunda influencia en la calidad del software y en el rendimiento como organización. La destreza y la motivación del personal se muestran como **el** Único factor realmente influyente en calidad y en el rendimiento [BOE81]. La complejidad del producto puede tener un impacto sustancial sobre la calidad y el rendimiento del equipo. La tecnología (por ejemplo: métodos de la ingeniería del software) que utiliza el proceso también tiene su impacto. Además, el triángulo de proceso existe dentro de un círculo de condiciones de entornos que incluyen entornos de desarrollo (por ejemplo: herramientas **CASE),** condiciones de gestión (por ejemplo: fechas tope, reglas de empresa) y características del cliente (por ejemplo: facilidad de comunicación).



¿Como puedo medir la efectividad de un proceso de software?

La eficacia de un proceso de software se mide indirectamente. Esto es, se extrae un juego de métricas según los resultados que provienen del proceso. Entre los resultados se incluyen medidas de errores detectados antes de la entrega del software, defectos detectados e informados a los usuarios finales, productos de trabajo entregados (productividad), esfuerzo humano y tiempo consumido, ajuste con la planificación y otras medidas. Las métricas de proceso también se extraen midiendo las características de tareas específicas de la ingeniería del software. Por ejemplo, se podría medir el tiempo y el esfuerzo de llevar a cabo las actividades de protección y las actividades genéricas de ingeniería del software del Capítulo 2.

Grady [GRA92] argumenta que existen unos usos «privados y públicos» para diferentes tipos de datos de proceso. Como es natural que los ingenieros del software pudieran sentirse sensibles ante la utilización de métricas recopiladas sobre una base particular, estos datos deberían ser *privados* para el individuo y servir sólo como un indicador de ese individuo. Entre los ejemplos de *métricas privadas* se incluyen: índices de defectos (individualmente), índices de defectos (por módulo), errores encontrados durante el desarrollo.

La filosofía de «datos de proceso privados» se ajusta bien con el enfoque del *proceso personal del software* propuesto por Humphrey [HUM95]. Humphrey describe el enfoque de la manera siguiente:

> El proceso personal del software (PPS) es un conjunto estructurado de descripciones de proceso, mediciones y métodos que pueden ayudar a que **los** ingenieros mejoren **su** rendimiento personal. Proporcionan las formas, guiones y estándares que les ayudan a estimar y planificar **su** trabajo. Muestra cómo definir procesos y cómo medir **su** calidad y productividad. Un principio PPS fundamental es que todo el mundo es diferente y que un método que sea efectivo para un ingeniero puede que no sea adecuado para otro. **Así** pues, el PPS ayuda a que **los** ingenieros midan y sigan la pista de su trabajo para que puedan encontrar **los** métodos que sean mejores para ellos.

Humphrey reconoce que la mejora del proceso del software puede y debe empezar en el nivel individual. Los datos privados de proceso pueden servir como referencia importante para mejorar el trabajo individual del ingeniero del software.

Algunas métricas de proceso son privadas para el equipo del proyecto de software, pero *públicas* para todos los miembros del equipo. Entre los ejemplos se incluyen los defectos informados de funciones importantes del software (que un grupo de profesionales han desarrollado), errores encontrados durante revisiones técnicas formales, y líneas de código o puntos de función por módulo y función[2]. El equipo revisa estos datos para detectar los indicadores que pueden mejorar el rendimiento del equipo.

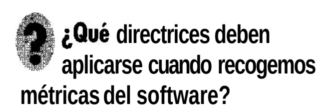


las métricas públicas permiten a una organización realizar cambios estratégicos que mejoran el proceso del software y cambios tácticos durante un proyecto de software.

Las métricas públicas generalmente asimilan información que originalmente era privada de particulares y equipos. Los índices de defectos a nivel de proyecto (no atribuidos absolutamente a un particular), esfuerzo, tiempo y datos afines se recopilan y se evalúan en un intento de detectar indicadores que puedan mejorar el rendimiento del proceso organizativo.

Las métricas del proceso del software pueden proporcionar beneficios significativos a medida que una organización trabaja por mejorar su nivel global de madurez del proceso. Sin embargo, al igual que todas las métricas, éstas pueden usarse mal, originando más problemas de los que pueden solucionar. Grady [GRA92] sugiere una «etiqueta de métricas del software» adecuada para gestores al tiempo que instituyen un programa de métricas de proceso:

- Utilice el sentido común y una sensibilidad organizativa al interpretar datos de métricas.
- Proporcione una retroalimentación regular para particulares y equipos que hayan trabajado en la recopilación de medidas y métricas.
- No utilice métricas para evaluar a particulares.
- Trabaje con profesionales y equipos para establecer objetivos claros y métricas que se vayan a utilizar para alcanzarlos.
- No utilice nunca métricas que amenacen a particulares o equipos.
- Los datos de métricas que indican un área de problemas no se deberían considerar «negativos». Estos datos son meramente un indicador de mejora de proceso.
- No se obsesione con una sola métrica y excluya otras métricas importantes.


¿Qué directrices deben aplicarse cuando recogemos métricas del software?

A medida que una organización está más a gusto con la recopilación y utiliza métricas de proceso, la derivación de indicadores simples abre el camino hacia un enfoque más riguroso llamado *mejora estadística de proceso del software (MEPS)*. En esencia, MEPS utiliza el análisis de fallos del software para recopilar infor-

[2] Consulte las Secciones 4.3.1 y 4.3.2 para estudios más detallados sobre LDC (líneas de código) y métricas de puntos de función.

mación de errores y defectos[3] encontrados al desarrollar y utilizar una aplicación de sistema o producto. El análisis de fallos funciona de la misma manera:



Referencia Web

MPSE y otra información relacionada con la calidad está disponible en la Sociedad Americana para la Calidad en **www.asq.org**

1. Todos los errores y defectos se categorizan por origen (por ejemplo: defectos en la especificación, en la lógica, no acorde con los estándares).
2. Se registra tanto el coste de corregir cada error como el del defecto.
3. El número de errores y de defectos de cada categoría se cuentan y se ordenan en orden descendente.
4. Se computa el coste global de errores y defectos de cada categoría.
5. Los datos resultantes se analizan para detectar las categorías que producen el coste más alto para la organización.
6. Se desarrollan planes para modificar el proceso con el intento de eliminar (o reducir la frecuencia de apariciones de) la clase de errores y defectos que sean más costosos.

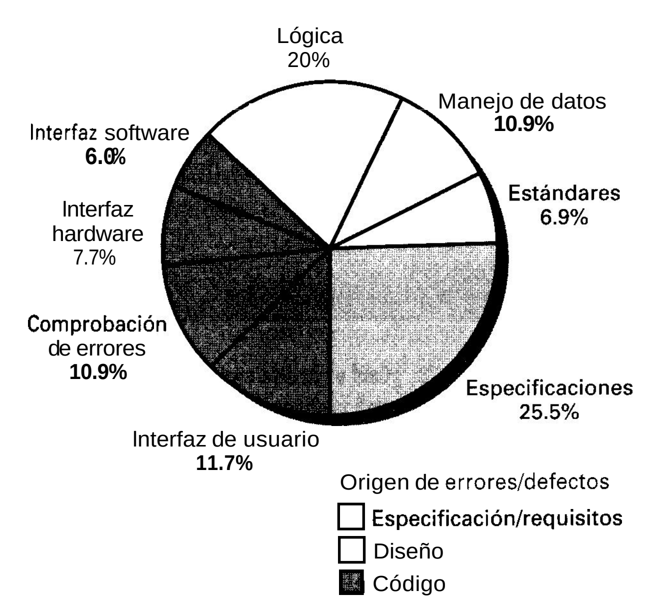


FIGURA 4.2. Causas de defectos y su origen para cuatro proyectos de software [GRA94].



CONSEJO

No puedes mejorar tu enfoque para la ingeniería del software a menos que comprendas dónde estás fuerte y dónde estás débil. Utilice las técnicas MEPS para aumentar esa comprensión.

Siguiendo los pasos 1 y 2 anteriores, se puede desarrollar una simple distribución de defectos (Fig. 4.2) [GRA94]. Para el diagrama de tarta señalado en la figura, se muestran ocho causas de defectos y sus ongenes (indicados en sombreado). Grady sugiere el desarrollo de un *diagrama de espina* [GRA92] para ayudar a diagnosticar los datos presentados en el diagrama de frecuencias. En la Figura **4.3** la espina del diagrama (la línea central) representa el factor de calidad en consideración (en este caso, los *defectos de especificación* que cuentan con el 25 por 100 del total). Cada una de las varillas (líneas diagonales) conectadas a la espina central indican una causa potencial del problema de calidad (por ejemplo: requisitos perdidos, especificación ambigua, requisitos incorrectos y requisitos cambiados). La notación de la espina y de las varillas se añade entonces a cada una de las varillas principales del diagrama para centrarse sobre la causa destacada. La expansión se muestra sólo para la causa «incorrecta» en la Figura **4.3.**

La colección de métricas del proceso es el conductor para la creación del diagrama en espina. Un diagrama en espina completo se puede analizar para extraer los indicadores que permitan a una organización de software modificar su proceso para reducir la frecuencia de errores y defectos.

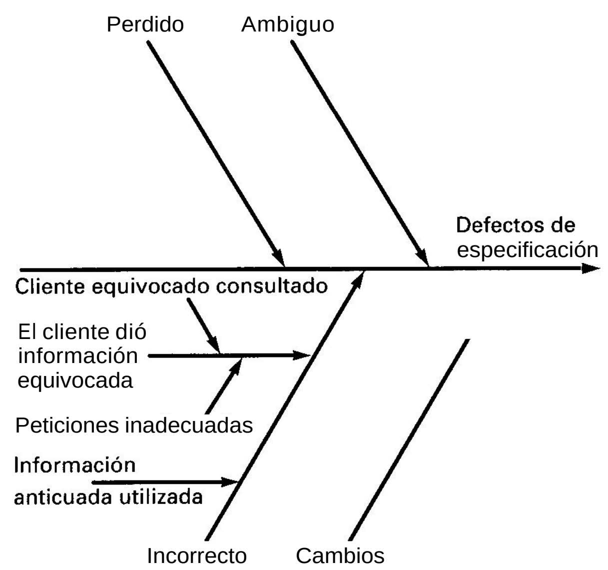


FIGURA 4.3. Un diagrama de espina (Adaptado de [GRA92]).

[3] Como se trata en el Capítulo 8, un *error* es alguna fisura en un producto de trabajo de ingeniería del software o en la entrega descubierta por los ingenieros del software antes de que el software sea entregado al usuario final.
Un defecto es una fisura descubierta después de la entrega al usuario final.

4.2.2. Métricas del proyecto

Las métricas del proceso de software se utilizan para propósitos estratégicos. Las medidas del proyecto de software son tácticas. Esto es, las métricas de proyectos y los indicadores derivados de ellos los utilizan un gestor de proyectos y un equipo de software para adaptar el flujo del trabajo del proyecto y las actividades técnicas.

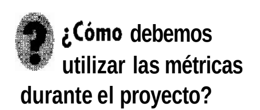
Referencia cruzada

las técnicas de estimación de proyectos se estudian en el capítulo 5.

La primera aplicación de métricas de proyectos en la mayoría de los proyectos de software ocurre durante la estimación. Las métricas recopiladas de proyectos anteriores se utilizan como una base desde la que se realizan las estimaciones del esfuerzo y del tiempo para el actual trabajo del software. **A** medida que avanza un proyecto, las medidas del esfuerzo y del tiempo consumido se comparan con las estimaciones originales (y la planificación de proyectos). El gestor de proyectos utiliza estos datos para supervisar y controlar el avance.

A medida que comienza el trabajo técnico, otras métricas de proyectos comienzan a tener significado. Se miden los índices de producción representados mediante páginas de documentación, las horas de revisión, los puntos de función y las líneas fuente entregadas. Además, se sigue la pista de los errores detectados durante todas las tareas de ingeniería del software. Cuando va evolucionando el software desde la especificación al diseño, se recopilan las métricas técnicas (Capítulos 19 al **24)** para evaluar la calidad del diseño y para proporcionar indicadores que influirán en el enfoque tomado para la generación y prueba del código.

La utilización de métricas para el proyecto tiene dos aspectos fundamentales. En primer lugar, estas métricas se utilizan para minimizar la planificación de desarrollo haciendo los ajustes necesarios que eviten retrasos y reduzcan problemas y riesgos potenciales. En segundo lugar, las métricas para el proyecto se utilizan para evaluar la calidad de los productos en el momento actual y cuando sea necesario, modificando el enfoque técnico que mejore la calidad.



A medida que mejora la calidad, se minimizan los defectos, y al tiempo que disminuye el número de defectos, la cantidad de trabajo que ha de rehacerse también se reduce. Esto lleva a una reducción del coste global del proyecto.

Otro modelo de métricas del proyecto de software [HET93] sugiere que todos los proyectos deberían medir:

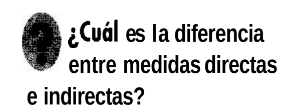
- entradas: la dimensión de los recursos (p. ej.: personas, entorno) que se requieren para realizar el trabajo,
- salidas: medidas de las entregas o productos creados durante el proceso de ingeniería del software,
- resultados: medidas que indican la efectividad de las entregas.

En realidad, este modelo se puede aplicar tanto al proceso como al proyecto. En el contexto del proyecto, el modelo se puede aplicar recursivamente a medida que aparece cada actividad estructural. Por consiguiente las salidas de una actividad se convierten en las entradas de la siguiente. Las métricas de resultados se pueden utilizar para proporcionar una indicación de la utilidad de los productos de trabajo cuando fluyen de una actividad (o tarea) a la siguiente.

4.3 MEDICIONES DEL SOFTWARE

Las mediciones del mundo físico se pueden categorizar de dos maneras; *medidas directas* (por ejemplo: la longitud de un tomillo) y *medidas indirectas* (por ejemplo: la «calidad» de los tomillos producidos, medidos contando los artículos defectuosos). Las métricas del software se pueden categorizar de forma similar.

Entre las medidas directas del proceso de la ingeniería del software se incluyen el coste y el esfuerzo aplicados. Entre las medidas directas del producto se incluyen las líneas de código (LDC) producidas, velocidad de ejecución, tamaño de memoria, y los defectos informados durante un período de tiempo establecido. Entre las medidas indirectas se incluyen la funcionalidad, calidad, complejidad, eficiencia, fiabilidad, facilidad de mantenimiento y muchas otras «capacidades» tratadas en el Capítulo 19.



El coste y el esfuerzo requerido para construir el software, el número de líneas de código producidas, y otras medidas directas son relativamente fáciles de reunir, mientras que los convenios específicos para la medición se establecen más adelante. Sin embargo, la calidad y funcionalidad del software, o su eficiencia o mantenimiento son más difíciles de evaluar y sólo pueden ser medidas indirectamente.

El dominio de las métricas del software se dividen en métricas de proceso, proyecto y producto. También se acaba de destacar que las métricas de producto que

son privadas para un individuo a menudo se combinan para desarrollar métricas del proyecto que sean públicas para un equipo de software. Las métricas del proyecto se consolidan para crear métricas de proceso que sean públicas para toda la organización del software. Pero jcómo combina una organización métricas que provengan de particulares o proyectos?

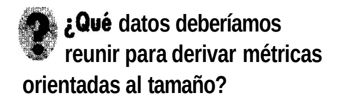


Puesto que muchos factores influyen en el trabajo del software, no utilice las métricas poro comparar personas o equipos.

Para ilustrarlo, se va a tener en consideración un ejemplo sencillo. Personas de dos equipos de proyecto diferentes registran y categorizan todos los errores que se encuentran durante el proceso del software. Las medidas de las personas se combinan entonces para desarrollar medidas de equipo. El equipo A encontró 342 errores durante el proceso del software antes de su realización. El equipo B encontró 184. Considerando que en el resto los equipos son iguales, ¿qué equipo es más efectivo en detectar errores durante el proceso'? Como no se conoce el tamaño o la complejidad de los proyectos, no se puede responder a esta pregunta. Sin embargo, si se normalizan las medidas, es posible crear métricas de software que permitan comparar medidas más amplias de otras organizaciones.

4.3.1. Métricas Orientadas al Tamaño

Las métricas del software orientadas al tamaño provienen de la normalización de las medidas de calidad y/o productividad considerando el «tamaño» del software que se haya producido. Si una organización de software mantiene registros sencillos, se puede crear una tabla de datos orientados al tamaño, como la que muestra la Figura 4.4. La tabla lista cada proyecto de desarrollo de software de los últimos años y las medidas correspondientes de cada proyecto. Refiriéndonos a la entrada de la tabla (Fig. 4.4) del proyecto *alfa:* se desarrollaron 12.100 líneas de código (LDC) con 24 personas-mes y con un coste de E168.000. Debe tenerse en cuenta que el esfuerzo y el coste registrados en la tabla incluyen todas las actividades de ingeniería del software (análisis, diseño, codificación y prueba) y no sólo la codificación. Otra información sobre el proyecto alfa indica que se desarrollaron 365 páginas de documentación, se registraron 134 errores antes de que el software se entregara y se encontraron 29 errores después de entregárselo al cliente dentro del primer año de utilización. También sabemos que trabajaron tres personas en el desarrollo del proyecto alfa.

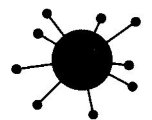


Proyecto	LDC	Esfuerzo	Coste £(000)	Pag. Doc.	Errores	Defectos	Personas
Alfa	12,100	24	168	365	134	29	3
Beta	27,200	62	440	1224	321	86	5
Gamma	20,200	43	314	1050	256	64	6
·		·	·	·	·		

FIGURA 4.4. Métricas orientadas al tamaño.

Para desarrollar métricas que se puedan comparar entre distintos proyectos, se seleccionan las líneas de código como valor de normalización. Con los rudimentarios datos contenidos en la tabla se pueden desarrollar para cada proyecto un conjunto de métricas simples orientadas al tamaño:

- errores por KLDC (miles de líneas de código)
- defectos[4] por KLDC
- E por LDC
- páginas de documentación por KLDC



Plantilla de colección de métricos

Además, se pueden calcular otras métricas interesantes:

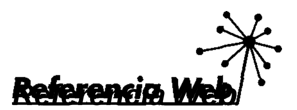
- errores por persona-mes
- LDC por persona-mes
- £ por página de documentación

4.3.2. Métricas Orientadas a la Función

Las métricas del software orientadas a la función utilizan una medida de la funcionalidad entregada por la aplicación como un valor de normalización. Ya que la «funcionalidad» no se puede medir directamente, se debe derivar indirectamente mediante otras medidas directas. Las métricas orientadas a la función fueron propuestas por primera vez por Albretch [ALB79], quien sugirió una medida llamada *punto defunción.* Los puntos de función se derivan con una relación empírica

[4] Un defecto ocurre cuando las actividades de garantía de calidad (p. ej.: revisiones técnicas formales) fallan para descubrir un error en un producto de trabajo generado durante el proceso del software.

según las medidas contables (directas) del dominio de información del software y las evaluaciones de la complejidad del software.



Se puede obtener información completo sobre los puntos de función en: **www.ifpug.org**

Los puntos de función se calculan [IFP94] completando la tabla de la Figura 4.5. Se determinan cinco características de dominios de información y se proporcionan las cuentas en la posición apropiada de la tabla. Los valores de los dominios de información se definen de la forma siguiente?

Número de entradas de usuario. Se cuenta cada entrada de usuario que proporciona diferentes datos orientados a la aplicación. Las entradas se deberían diferenciar de las peticiones, las cuales se cuentan de forma separada.



los puntos de función se derivan de medidas directas del dominio de la información.

Número de salidas de usuario. Se cuenta cada salida que proporciona al usuario información orientada a la aplicación. En este contexto la salida se refiere a informes, pantallas, mensajes de error, etc. Los elementos de datos particulares dentro de un informe no se cuentan de forma separada.

Número de peticiones de usuario. Una petición se define como una entrada interactiva que produce la generación de alguna respuesta del software inmediata en forma de salida interactiva. Se cuenta cada petición por separado.

Número de archivos. Se cuenta cada archivo maestro lógico (esto es, un grupo lógico de datos que puede ser una parte de una gran base de datos o un archivo independiente).

Número de interfaces externas. Se cuentan todas las interfaces legibles por la máquina (por ejemplo: archivos de datos de cinta o disco) que se utilizan para transmitir información a otro sistema.

Una vez que se han recopilado los datos anteriores, a la cuenta se asocia un valor de complejidad. Las organizaciones que utilizan métodos de puntos de función desarrollan criterios para determinar si una entrada en particular es simple, media o compleja. No obstante la determinación de la complejidad es algo subjetiva.

Parámetros de medición	Cuenta		Factor de ponderación: Simple	Medio	Complejo		
Número de entradas de usuario	☐	x	3	4	6	=	☐
Número de salidas de usuario	☐	x	4	5	7	=	☐
Número de peticiones de usuario	☐	x	3	4	6	=	☐
Número de archivos	☐	x	7	10	15	=	☐
Número de interfaces externas	☐	x	5	7	10	=	☐
Cuenta total	→						☐

FIGURA 4.5. Cálculo de puntos de función.

Para calcular puntos de función (PF), se utiliza la relación siguiente:

$$PF = \text{cuenta-total} \times [0{,}65 + 0{,}01 \times \Sigma(F_i)] \quad (4.1)$$

en donde cuenta-total es la suma de todas las entradas PF obtenidas de la figura 4.5.

F_i (i = 1 a 14) son «valores de ajuste de la complejidad» según las respuestas a las siguientes preguntas [ART85]:

1. ¿Requiere el sistema copias de seguridad y de recuperación fiables?
2. ¿Se requiere comunicación de datos?
3. ¿Existen funciones de procesamiento distribuido?
4. ¿Es crítico el rendimiento?
5. ¿Se ejecutará el sistema en un entorno operativo existente y fuertemente utilizado?
6. ¿Requiere el sistema entrada de datos interactiva?
7. ¿Requiere la entrada de datos interactiva que las transacciones de entrada se lleven a cabo sobre múltiples pantallas u operaciones?
8. ¿Se actualizan los archivos maestros de forma interactiva?
9. ¿Son complejas las entradas, las salidas, los archivos o las peticiones?
10. ¿Es complejo el procesamiento interno?
11. ¿Se ha diseñado el código para ser reutilizable?
12. ¿Están incluidas en el diseño la conversión y la instalación'?
13. ¿Se ha diseñado el sistema para soportar múltiples instalaciones en diferentes organizaciones?
14. ¿Se ha diseñado la aplicación para facilitar los cambios y para ser fácilmente utilizada por el usuario?

[5] En realidad, la definición de los valores del dominio de la información y la forma en que se cuentan es un poco más complejo. El lector interesado debería consultar [IFP94] para obtener más detalles.

Cada una de las preguntas anteriores es respondida usando una escala con rangos desde 0 (no importante o aplicable) hasta 5 (absolutamente esencial). Los valores constantes de la ecuación (4.1) y los factores de peso que se aplican a las cuentas de los dominios de información se determinan empíricamente.

Una vez que se han calculado los puntos de función, se utilizan de forma análoga a las LDC como forma de normalizar las medidas de productividad, calidad y *otros* atributos del software.

4.3.3. Métricas ampliadas de punto de función

La medida de punto de función se diseñó originalmente para aplicarse a aplicaciones de sistemas de información de gestión. Para acomodar estas aplicaciones, se enfatizó la dimensión de datos (los valores de dominios de información tratados anteriormente) para la exclusión de dimensiones (control) funcionales y de comportamiento. Por esta razón, la medida del punto de función era inadecuada para muchos sistemas de ingeniería y sistemas empotrados (que enfatizan función y control). Para remediar esta situación se ha propuesto un número de extensiones a la métrica del punto de función básica.



La extensión de los puntos de función se utiliza en la ingeniería, en las aplicaciones de tiempo real y en las aplicaciones orientadas al control.

Una extensión del punto de función es la llamada puntos de características [JON91]; es una ampliación de la medida del punto de función que se puede aplicar a sistemas y aplicaciones de ingeniería del software. La medida de punto de característica acomoda a aplicaciones en donde la complejidad del algoritmo es alta. Las aplicaciones de software de tiempo real, de control de procesos, y empotradas tienden a tener alta complejidad de algoritmos y por lo tanto son adecuadas para el punto de característica.

Para calcular el punto de característica, los valores de dominio de información se cuentan otra vez y se pesan de la forma que se describe en la Sección 4.3.2. Además, la métrica del punto de característica cuenta una característica nueva del software —los *algoritmos*—. Un algoritmo se define como «un problema de cálculo limitado que se incluye dentro de un programa de computadora específico» [JON91]. Invertir una matriz, decodificar una cadena de bits o manejar una interrupción son ejemplos de algoritmos.



Se puede encontrar una lista útil de preguntas realizadas con frecuencia sobre los puntos de función (y puntos de función extendidos) en:
http://ourworld.compuserve.com/homepages/softcomp/

Boeing ha desarrollado otra extensión de punto de función para sistemas en tiempo real y productos de ingeniería. El enfoque de Boeing integra la dimensión de datos del software con las dimensiones funcionales y de control para proporcionar una medida orientada a la función que es adecuada a aplicaciones que enfatizan las capacidades de control y función. Las características de las tres dimensiones del software se «cuentan, cuantifican y transforman» en una medida que proporciona una indicación de la funcionalidad entregada por el software[6], llamada *Punto de Función 3D* [WHI95]

La *dimensión de datos* se evalúa exactamente igual a como se describe en la Sección 4.3.2. Las cuentas de datos retenidos (la estructura interna de datos de programas, p. ej.: archivos) y los datos externos (entradas, salidas, peticiones y referencias externas) se utilizan a lo largo de las medidas de la complejidad para derivar una cuenta de dimensión de datos.

La *dimensión funcional* se mide considerando «el número de operaciones internas requeridas para transformar datos de entrada en datos de salida» [WHI95]. Para propósitos de computación de los puntos de función 3D, se observa una «transformación» como una serie de pasos de proceso que quedan limitados por sentencias semánticas. La *dimensión de control* se mide contando el número de transiciones entre estados[7].

Un estado representa algún modo de comportamiento externamente observable, y una transición ocurre como resultado de algún suceso que cambia el modo de comportamiento del sistema o del software (esto es, cambia el estado). Por ejemplo, un teléfono inalámbrico contiene software que soporta funciones de marcado automático. Para introducir el estado de marcado automático desde un estado en descanso, el usuario pulsa una tecla **Auto** en el teclado numérico. Este suceso hace que una pantalla LCD pida un código que indique la parte que se llama. Con la entrada del código y pulsando la **tecla de Marcado** (otro suceso), el software del teléfono inalámbrico hace una transición al estado de marcado. Cuando se computan puntos de función 3D, no se asigna un valor de complejidad a las transiciones.

[6] Debe señalarse que también han sido propuestas otras extensiones a los puntos de función para aplicarlas en trabajos de software de tiempo real (p.ej., [ALA97]). Sin embargo, ninguno de estos parece usarse ampliamente en la industria.

[7] En el capítulo 12 se presenta un estudio detallado de la dimensión de comportamientos que incluyen estados y transiciones de estados.

Para calcular puntos de función 3D, se utiliza la relación siguiente:

$$\text{índice} = I + O + Q + F + E + T + R \qquad (4.2)$$

en donde I, O, Q, F, E, T y R representan valores con peso de complejidad en los elementos tratados anteriormente: entradas, salidas, peticiones, estructuras de datos internas, archivos externos, transformaciones y transiciones, respectivamente. Cada valor con peso de complejidad se calcula con la relación siguiente:

valor con peso de complejidad =

$$= N_{il} W_{il} + N_{ia} W_{ia} + N_{ih} W_{ih} \qquad (4.3)$$

en donde N_{il}, N_{ia} y N_{ih} representan el número de apariciones del elemento i (p. ej.: salidas) para cada nivel de complejidad (bajo, medio, alto), y W_{il}, W_{ia} y W_{ih} son los pesos correspondientes. El cálculo global de los puntos de función 3D se muestran en la Figura 4.6.

Se debería señalar que los puntos de función, los puntos de característica y los puntos de función 3D representan lo mismo —«funcionalidad» o «utilidad» entregada por el software—. En realidad, cada una de estas medidas producen el mismo valor si sólo se considera la dimensión de datos de una aplicación. Para sistemas de tiempo real más complejos, la cuenta de puntos de característica a menudo se encuentra entre el 20 y el 35 por 100 más que la cuenta determinada sólo con los puntos de función.

El punto de función (y sus extensiones), como la medida LDC, es controvertido. Los partidarios afirman que PF es independiente del lenguaje de programación, lo cual es ideal para aplicaciones que utilizan lenguajes convencionales y no procedimentales; esto se basa en los datos que probablemente se conocen al principio de la evolución de un proyecto, y así el PF es más atractivo como enfoque de estimación.

Pasos de proceso \ Sentencias semánticas	1–5	6–10	11+
1–10	Bajo	Bajo	Medio
11–20	Bajo	Medio	Alto
21+	Medio	Alto	Alto

FIGURA 4.6. Cálculo de los índices de los puntos de función 3D.

Los detractores afirman que el método requiere algún «juego de manos» en el que el cálculo se base en datos subjetivos y no objetivos; y afirman que las cuentas del dominio de información (y otras dimensiones) pueden ser difíciles de recopilar después de realizado, y por último que el PF no tiene un significado físico directo —es simplemente un número—.

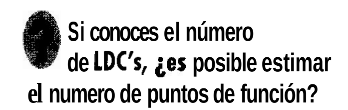
4.4 RECONCILIACIÓN DE LOS DIFERENTES ENFOQUES DE MÉTRICAS

La relación entre las líneas de código y los puntos de función depende del lenguaje de programación que se utilice para implementar el software, y de la calidad del diseño. Ciertos estudios han intentado relacionar las métricas LDC y PF. Citando a Albrecht y Gaffney [ALB83]:

> La tesis de este trabajo es que la cantidad de función proporcionada por la aplicación (programa) puede ser estimada por la descomposición de los principales componentes[8] de datos que usa o proporciona el programa. Además, esta estimación de la función debe ser relacionada con el total de LDC a desarrollar y con el esfuerzo de desarrollo necesario.

La tabla siguiente [JON98] proporciona estimaciones informales del número medio de líneas de código que se requiere para construir un punto de función en varios lenguajes de programación:

Lenguaje de programación	LDC/PF (media)
Ensamblador	320
C	128
COBOL	106
FORTRAN	106
Pascal	90
C++	64
Ada95	53
Visual Basic	32
Smalltalk	22
Powerbuilder (generador de código)	16
SQL	12





Utilice el repaso de los datos de un modo juicioso. Es bastante mejor calcular los PF utilizando los métodos utilizadas anteriormente.

[8] Es importante destacar que la «la individualización de componentes importantes» se puede interpretar de muchas maneras. Algunos ingenieros del software que trabajan en un entorno de desarrollo orientado a objetos (Cuarta Parte) utilizan ciertas clases u objetos como métrica dominante del tamaño. Una organización de mantenimiento podría observar el tamaño de los proyectos en relación con los pedidos de cambio de ingeniería (Capítulo 9). Una organización de sistemas de información podría observar el número de procesos de negocio a los que impacta una aplicación.

Una revisión de los datos anteriores indica que una LDC de C++ proporciona aproximadamente **1,6** veces la «funcionalidad» (de media) que una LDC de FORTRAN. Además, una LDC de Visual Basic proporciona ***3*** veces la funcionalidad de una LDC de un lenguaje de programación convencional. En [JON98] se presentan datos más precisos sobre las relaciones entre los PF y las LDC, que pueden ser usados para «repasar» programas existentes, determinando sus medidas en PF (por ejemplo, para calcular el número de puntos de función cuando se conoce la cantidad de LDC entregada).

Las medidas LDC y PF se utilizan a menudo para extraer métricas de productividad. Esta invariabilidad conduce al debate sobre el uso de tales datos. Se debe comparar la relación LDC/personas-mes (ó PF/personas-mes) de un grupo con los datos similares de otro grupo? ¿Deben los gestores evaluar el rendimiento de las personas usando estas métricas? La respuesta a estas preguntas es un rotundo «No». La razón para esta respuesta es que hay muchos factores que influyen en la productividad, haciendo que la comparación de «manzanas y naranjas» sea mal interpretada con facilidad.

Se han encontrado los puntos de función y las LDC basadas en métricas para ser predicciones relativamente exactas del esfuerzo y coste del desarrollo del software. Sin embargo, para utilizar LDC y PF en las técnicas de estimación (capítulo 5), debe establecerse una línea base de información histórica.

4.5 MÉTRICAS PARA LA CALIDAD DEL SOFTWARE

El objetivo primordial de la ingeniería del software es producir un sistema, aplicación o producto de alta calidad. Para lograr este objetivo, los ingenieros del software deben aplicar métodos efectivos junto con herramientas modernas dentro del contexto de un proceso maduro de desarrollo de software. Además, un buen ingeniero del software (y buenos gestores de la ingeniería del software) deben medir si la alta calidad se va a llevar a cabo.

La calidad de un sistema, aplicación o producto es tan bueno como los requisitos que describen el problema, el diseño que modela la solución, el código que conduce a un programa ejecutable, y las pruebas que ejercitan el software para detectar errores. Un buen ingeniero del software utiliza mediciones que evalúan la calidad del análisis y los modelos de diseño, el código fuente, y los casos de prueba que se han creado al aplicar la ingeniería del software. Para lograr esta evaluación de la calidad en tiempo real, el ingeniero debe utilizar *medidas técnicas* (Capítulos 19 y 24) que evalúan la calidad con objetividad, no con subjetividad.



Se puede encontrar una excelente fuente de información sobre la calidad del software y ternos relacionados (incluyendo métricas) en: **www.qualityworld.com**

El gestor de proyectos también debe evaluar la calidad objetivamente, y no subjetivamente. **A** medida que el proyecto progresa el gestor del proyecto también debe evaluar la calidad. Las métricas privadas recopiladas por ingenieros del software particulares se asimilan para proporcionar resultados en los proyectos. Aunque se pueden recopilar muchas medidas de calidad, el primer objetivo en el proyecto es medir errores y defectos. Las métricas que provienen de estas medidas proporcionan una indicación de la efectividad de las actividades de control y de la garantía de calidad en grupos o en particulares.

Referencia cruzada

En el capitulo 8 se presento un **estudio** detallada sobre **las** actividades de garantía de calidad del **software.**

4.5.1. Visión general de los factores que afectan a la calidad

Hace 25 años, McCall y Cavano [MCC78] definieron un juego de factores de calidad como los primeros pasos hacia el desarrollo de métricas de la calidad del software. Estos factores evalúan el software desde tres puntos de vista distintos: (1) operación del producto (utilizándolo), (2) revisión del producto (cambiándolo), y (3) transición del producto (modificándolo para que funcione en un entorno diferente, por ejemplo, «portándolo»). Los autores, en su trabajo, describen la relación entre estos factores de calidad (lo que llaman un «marco de trabajo») y otros aspectos del proceso de ingeniería del software:

> En primer lugar, el marco de trabajo proporciona un mecanismo para que el gestor del proyecto identifique lo que considera importante. Estas cualidades son atributos del software, además de su corrección y rendimiento funcional, que tiene implicaciones en el ciclo de vida. En otros factores, como son facilidad de mantenimiento y portabilidad, se ha demostrado que tienen un impacto significativo en el coste del ciclo de vida.. .
>
> En segundo lugar, el marco de trabajo proporciona un medio de evaluar cuantitativamente lo bien que va progresando el desarrollo en relación con **los** objetivos de calidad establecidos.. .
>
> En tercer lugar, el marco de trabajo proporciona más interacción del personal de **QA** (garantía de calidad) en el esfuerzo de desarrollo.. .
>
> Por Último,. .. el personal de garantía de calidad puede utilizar indicaciones de calidad pobre para ayudar a identificar estándares [mejores] a enfrentar en el futuro.

Un estudio detallado del marco de trabajo de McCall y Cavano, así como otros factores de calidad, se presentan en el Capítulo 19. Es interesante destacar que casi todos los aspectos del cálculo han sufrido cambios radicales con el paso de los años desde que McCall y

Cavano hicieron su trabajo, con gran influencia, en 1978. Pero los atributos que proporcionan una indicación de la calidad del software siguen siendo los mismos.



Sorprendentemente, los factores que definían la calidad del software en el año 1970 san los mismos factores que continúan definiendo la calidad del software en la primera década de este siglo.

¿Qué significa esto? Si una organización de software adopta un juego de factores de calidad como una «lista de comprobación» para evaluar la calidad del software, es probable que el software construido hoy siga exhibiendo la calidad dentro de las primeras décadas de este siglo. Incluso, cuando las arquitecturas de cálculo sufren cambios radicales (como seguramente ocurrirá), el software que exhibe alta calidad en operación, transición y revisión continuará sirviendo también a sus usuarios.

4.5.2. Medida de la calidad

Aunque hay muchas medidas de la calidad de software, la *corrección, facilidad de mantenimiento, integridad*, y *facilidad de uso* proporcionan indicadores Útiles para el equipo del proyecto. Gilb [GIL88] ha sugerido definiciones y medidas para cada uno de ellos.

Corrección. Un programa debe operar correctamente o proporcionará poco valor a sus usuarios. La corrección es el grado en el que el software lleva a cabo su función requerida. La medida más común de corrección es *defectos por KLDC*, en donde un defecto se define como una falta verificada de conformidad con los requisitos.

Facilidad de mantenimiento. El mantenimiento del software cuenta con más esfuerzo que cualquier otra actividad de ingeniería del software. La facilidad de mantenimiento es la facilidad con la que se puede corregir un programa si se encuentra un error, se puede adaptar si su entorno cambia, o mejorar si el cliente desea un cambio de requisitos. No hay forma de medir directamente la facilidad de mantenimiento; por consiguiente, se deben utilizar medidas indirectas. Una simple métrica orientada al tiempo es el *tiempo medio de cambio (TMC)*, es decir el tiempo que se tarda en analizar la petición de cambio, en diseñar una modificación adecuada, en implementar el cambio, en probarlo y en distribuir el cambio a todos los usuarios. Como media, los programas que se pueden mantener tendrán un TMC más bajo (para tipos equivalentes de cambios) que los programas que no son más fáciles de mantener.

Hitachi [TAJ81] ha utilizado una métrica orientada al coste para la capacidad de mantenimiento llamada «desperdicios» —el coste en corregir defectos encontrados después de haber distribuido el software a sus usuarios finales—. Cuando la proporción de desperdicios en el coste global del proyecto (para muchos proyectos) se representa como una función del tiempo, el gestor puede determinar si la facilidad total del software producido por una organización de desarrollo está mejorando. Se pueden emprender acciones a partir de las conclusiones obtenidas de esa información.

Integridad. En esta época de «hackers» y «firewalls», la integridad del software ha llegado a tener mucha importancia. Este atributo mide la capacidad de un sistema para resistir ataques (tanto accidentales como intencionados) contra su seguridad. El ataque se puede realizar en cualquiera de los tres componentes del software: programas, datos y documentos.

Para medir la integridad, se tienen que definir dos atributos adicionales: amenaza y seguridad. *Amenaza* es la probabilidad (que se puede estimar o deducir de la evidencia empírica) de que un ataque de un tipo determinado ocurra en un tiempo determinado. La *seguridad* es la probabilidad (que se puede estimar o deducir de la evidencia empírica) de que se pueda repeler el ataque de un tipo determinado. La integridad del sistema se puede definir como:

$$\text{integridad} = \sum [(1 - \text{amenaza}) \times (1 - \text{seguridad})]$$

donde se suman la amenaza y la seguridad para cada tipo de ataque.

Facilidad de uso. El calificativo «amigable con el usuario» se ha convertido en omnipresente en las discusiones sobre productos de software. Si un programa no es «amigable con el usuario», frecuentemente está abocado al fracaso, incluso aunque las funciones que realice sean valiosas. La facilidad de uso es un intento de cuantificar «lo amigable que puede ser con el usuario» y se puede medir en función de cuatro características: (1) habilidad intelectual y/o física requerida para aprender el sistema; (2) el tiempo requerido para llegar a ser moderadamente eficiente en el uso del sistema; *(3)* aumento neto en productividad (sobre el enfoque que el sistema reemplaza) medida cuando alguien utiliza el sistema moderadamente y eficientemente; y (4) valoración subjetiva (a veces obtenida mediante un cuestionario) de la disposición de los usuarios hacia el sistema. En el Capítulo 15 se estudia más detalladamente este aspecto.

Los cuatro factores anteriores son sólo un ejemplo de todos los que se han propuesto como medidas de la calidad del software. El Capítulo 19 considera este tema con más detalle.

4.5.3. Eficacia de la Eliminación de Defectos

Una métrica de la calidad que proporciona beneficios tanto a nivel del proyecto como del proceso, es la *eficacia de la eliminación de defectos (EED)*. En esen-

cia, EED es una medida de la habilidad de filtrar las actividades de la garantía de calidad y de control al aplicarse a todas las actividades del marco de trabajo del proceso.

Cuando un proyecto se toma en consideración globalmente, EED se define de la forma siguiente:

$$EED = E / (E + D) \quad (4.4)$$

donde E es el número de errores encontrados antes de la entrega del software al usuario final y D es el número de defectos encontrados después de la entrega.



El valor ideal de EED es 1. Esto es, no se han encontrado defectos en el software. De forma realista, ***D*** será mayor que cero, pero el valor de EED todavía se puede aproximar a 1. Cuando *E* aumenta (para un valor de D dado), el valor total de EED empieza a aproximarse a 1. De hecho, a medida que ***E*** aumenta, es probable que el valor final de D disminuya (los errores se filtran antes de que se conviertan en defectos). Si se utilizan como una métrica que proporciona un indicador de la habilidad de filtrar las actividades de la garantía de la calidad y del control, EED anima a que el equipo del proyecto de software instituya técnicas para encontrar todos los errores posibles antes de su entrega.

EED también se puede utilizar dentro del proyecto para evaluar la habilidad de un equipo en encontrar errores antes de que pasen a la siguiente actividad estructural o tarea de ingeniería del software. Por ejemplo, la tarea del análisis de los requisitos produce un modelo de **análisis** que se puede revisar para encontrar y corregir errores. Esos errores que no se encuentren durante la revisión del modelo de análisis se pasan a la tarea de diseño (en donde se pueden encontrar o no). Cuando se utilizan en este contexto, EED se vuelve a definir como:

$$EED_i = E_i / (E_i + E_{i+1}) \quad (4.5)$$

en donde E_i es el número de errores encontrado durante la actividad de ingeniería del software *i* y E_{i+1} es el número de errores encontrado durante la actividad de ingeniería del software $i + 1$ que se puede seguir para llegar a errores que no se detectaron en la actividad de la ingeniería del software *i*.



Utilice EED como una medida de la eficiencia de sus primeras actividades de ***SQA****. Si EED es* ***bajo*** *durante el análisis y diseño, emplee* ***algún*** *tiempo* ***mejorando*** *la forma de conducir las revisiones técnicas formales.*

4.6 INTEGRACIÓN DE LAS MÉTRICAS DENTRO DEL PROCESO DE INGENIERÍA DEL SOFTWARE

La mayoría de los desarrolladores de software todavía no miden, y por desgracia, la mayoría no desean ni comenzar. Como se ha señalado en este capítulo, el problema es cultural. En un intento por recopilar medidas en donde no se haya recopilado nada anteriormente, a menudo se opone resistencia: «¿Por qué necesitamos hacer esto?», se pregunta un gestor de proyectos agobiado. «No entiendo por qué», se queja un profesional saturado de trabajo.

En esta sección, consideraremos algunos argumentos para las métricas del software y presentaremos un enfoque para instituir un programa de métricas dentro de una organización de ingeniería del software. Pero antes de empezar, consideremos algunas palabras de cordura sugeridas por Grady y Caswell [GRA87]:

> Algunas de las cosas que describimos aquí parecen bastante fáciles. Realmente, sin embargo, establecer un programa de métricas del software con éxito es un trabajo duro. Cuando decimos esto debes esperar al menos tres años antes de que estén disponibles las tendencias organizativas, lo que puede darte alguna idea del ámbito de tal esfuerzo.

La advertencia sugerida por los autores se considera de un gran valor, pero los beneficios de la medición son tan convincentes que obligan a realizar este duro trabajo.

4.6.1. Argumentos para las Métricas del Software

¿Por qué es tan importante medir el proceso de ingeniería del software y el producto (software) que produce? La respuesta es relativamente obvia. Si no se mide, no hay una forma real de determinar si se está mejorando. Y si no se está mejorando, se está perdido.



Manejamos cosas «con los números» en muchos aspectos de nuestras vidas... Estos números nos dan una visión y nos ayudan a dirigir nuestras acciones.
Michael Mah
Larry Putnam

La gestión de alto nivel puede establecer objetivos significativos de mejora del proceso de ingeniería del software solicitando y evaluando las medidas de productividad y de calidad. En el Capítulo1 se señaló que el software es un aspecto de gestión estratégico para muchas compañías. Si el proceso por el que se desarrolla puede ser mejorado, se producirá un impacto directo en lo sustancial. Pero para establecer objetivos de mejora, se debe comprender el estado actual de desa-

rrollo del software. Por lo tanto la medición se utiliza para establecer una línea base del proceso desde donde se pueden evaluar las mejoras.

Los rigores del trabajo diario de un proyecto del software no dejan mucho tiempo para pensar en estrategias. Los gestores de proyecto de software están más preocupados por aspectos mundanos (aunque igualmente importantes): desarrollo de estimaciones significativas del proyecto; producción de sistemas de alta calidad y terminar el producto a tiempo. Mediante el uso de medición para establecer una línea base del proyecto, cada uno de estos asuntos se hace más fácil de manejar. Ya hemos apuntado que la línea base sirve como base para la estimación. Además, la recopilación de métricas de calidad permite a una organización «sintonizar» su proceso de ingeniería del software para eliminar las causas «poco vitales» de los defectos que tienen el mayor impacto en el desarrollo del software'.

Técnicamente (en el fondo), las métricas del software, cuando se aplican al producto, proporcionan beneficios inmediatos. Cuando se ha terminado el diseño del software, la mayoría de los que desarrollan pueden estar ansiosos por obtener respuestas a preguntas como:

- ¿Qué requisitos del usuario son más susceptibles al cambio?
- ¿Qué módulos del sistema son más propensos a error?
- ¿Cómo se debe planificar la prueba para cada módulo?
- ¿Cuántos errores (de tipos concretos) puede esperar cuando comience la prueba?

Se pueden encontrar respuestas a esas preguntas si se han recopilado métricas y se han usado como guía técnica. En posteriores capítulos examinaremos cómo se hace esto.

4.6.2. Establecimiento de una Línea Base

Estableciendo una línea base de métricas se pueden obtener beneficios a nivel de proceso, proyecto y producto (técnico). Sin embargo la información reunida no necesita ser fundamentalmente diferente. Las mismas métricas pueden servir varias veces. Las líneas base de métricas constan de datos recogidos de proyectos de software desarrollados anteriormente y pueden ser tan simples como la tabla mostrada en la figura **4.4** o tan complejas como una gran base de datos que contenga docenas de medidas de proyectos y las métricas derivadas de ellos.

Para ser una ayuda efectiva en la mejora del proceso y/o en la estimación del esfuerzo y del coste, los datos de línea base deben tener los siguientes atributos: (1) los datos deben ser razonablemente exactos —se deben evitar «conjeturas» sobre proyectos pasados—; (2) los datos deben reunirse del mayor número de proyectos que sea posible; *(3)* las medidas deben ser consistentes, por ejemplo, una línea de código debe interpretarse consistentemente en todos los proyectos para los que se han reunido los datos: **(4)** las aplicaciones deben ser semejantes para trabajar en la estimación —tiene poco sentido utilizar una línea base obtenida en un trabajo de sistemas de información batch para estimar una aplicación empotrada de tiempo real—.

4.6.3. Colección de métricas, cómputo y evaluación

El proceso que establece una línea base se muestra en la Figura 4.7. Idealmente, los datos necesarios para establecer una línea base se han ido recopilando a medida que se ha ido progresando. Por desgracia, este no es el caso. Por consiguiente, la recopilación de datos requiere una investigación histórica de los proyectos anteriores para reconstruir los datos requeridos. Una vez que se han recopilado medidas (incuestionablemente el paso más difícil), el cálculo de métricas es posible. Dependiendo de la amplitud de las medidas recopiladas, las métricas pueden abarcar una gran gama de métricas LDC y PF, así como métricas de la calidad y orientadas al proyecto. Finalmente, las métricas se deben evaluar y aplicar durante la estimación, el trabajo técnico, el control de proyectos y la mejora del proceso. La evaluación de métricas se centra en las razones de los resultados obtenidos, y produce un grupo de indicadores que guían el proyecto o el proceso.



los datos de las métricas de línea base deberían recogerse de una gran muestra representativa de proyectos de software del pasado.

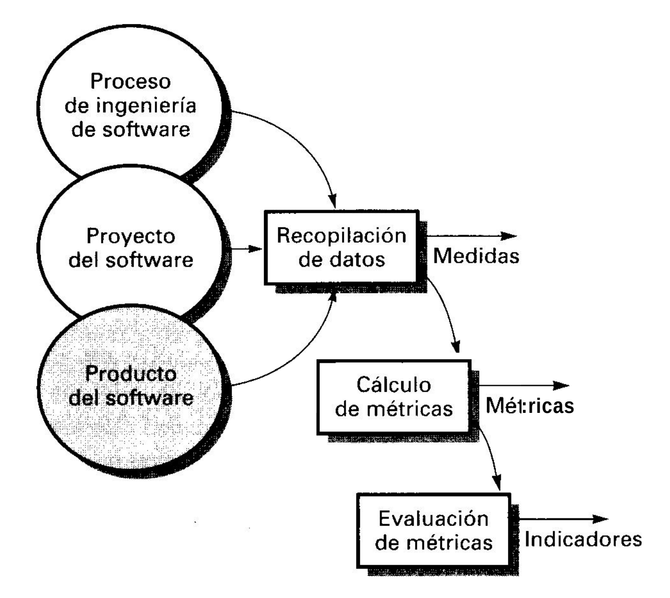


FIGURA 4.7. Proceso de recopilación de métricas del software.

[9] Estas ideas se han formalizado en un enfoque denominado *garantía estadística de calidad del software* y se estudian en detalle en el Capítulo 8.

4.7 EL DESARROLLO DE LA MÉTRICA Y DE LA OPM (Objetivo, Pregunta, Métrica)

Uno de los problemas al que se han enfrentado los trabajadores de las métricas durante las dos últimas décadas es la de desarrollar métricas que fueran útiles para el diseñador de software. Ha sido toda una historia de utilización de la métrica dentro de los entornos industriales basada en el simple criterio de lo que era facilitar la medida, más que emplear cualquier criterio relacionado con la utilidad. El panorama de la Última mitad de los años 80 y la primera mitad de la década de los 90, constató el hecho de que mientras había sido desarrollado mucho trabajo en la validación de la métrica y en el esclarecimiento de los principios teóricos detrás de ella, muy poco había sido hecho para dotar al diseñador de software con herramientas para la selección o construcción de métricas.

El objetivo de esta sección es describir OPM, que es casi con certeza el método de desarrollo de métrica más ampliamente aplicado y mejor conocido y que ha sido desarrollado por Victor Basili y sus colaboradores de la Universidad de Maryland. Basili y sus colaboradores tenían ya una larga historia de validación de métricas en la década de los 80 y el método OPM (objetivo, pregunta y métrica) surgió de un trabajo que fue desarrollado dentro de un laboratorio de ingeniería del software esponsorizado por la Agencia Americana del Espacio, NASA.

Basili establecía que para que una organización tuviera un programa de medida exacto era necesario que tuviera constancia de tres componentes:

1. Un proceso donde pudieran articularse metas u objetivos para sus proyectos.
2. Un proceso donde estas metas pudieran ser traducidas a los datos del proyecto que exactamente reflejasen dichas metas u objetivos en términos de software.
3. Un proceso que interpretara los datos del proyecto con el fin de entender los objetivos.

Un ejemplo de un objetivo típico que bien podría ser adoptado por una empresa es que la cantidad de trabajo de revisión sobre un diseño de sistema debido a problemas que fueran descubiertos por los programadores se redujera al 80 por ciento.

A partir de este ejemplo de objetivo emerge un cierto número de preguntas típicas cuya contestación podría ser necesaria para clarificar los objetivos y por consiguiente el desarrollo de una métrica. Un conjunto de ejemplos de estas preguntas, se exponen a continuación:

- ¿Cómo realizar la cuantificación de los trabajos de revisión?
- ¿Debería ser tenido en cuenta el tamaño del producto en el cálculo de la disminución de trabajos de revisión o revisiones?
- Debería ser tenido en cuenta el incremento de mano de obra de programación requerida para validaciones suplementarias y trabajos de rediseño en comparación con el proceso actual de validar un diseño corriente.

Una vez que estas cuestiones se hayan establecido, entonces es cuando puede ser desarrollada una métrica que pueda ayudar a que se cumpla el objetivo planteado que naturalmente emergerá a partir de estas cuestiones.

La importancia de OPM proviene no solamente del hecho de que es uno de los primeros intentos de desarrollar un conjunto de medidas adecuado que pueda ser aplicado al software, sino también al hecho de que está relacionado con el paradigma de mejora de procesos que ha sido discutido previamente. Basili ha desarrollado un paradigma de mejora de calidad dentro del cual el método OPM puede encajarse perfectamente. El paradigma comprende tres etapas:

- Un proceso de llevar a cabo una auditoría de un proyecto y su entorno estableciendo metas u objetivos de calidad para el proyecto y seleccionando herramientas adecuadas y métodos y tecnologías de gestión para que dichos objetivos de calidad tengan una oportunidad de cumplirse.
- Un proceso de ejecutar un proyecto y chequear los datos relacionados con esas metas u objetivos de calidad. Este proceso es llevado a cabo en conjunción con otro proceso paralelo de actuación sobre los propios datos cuando se vea que el proyecto corre peligro de no cumplir con los objetivos de calidad.
- Un proceso para el análisis de los datos del segundo paso, con el fin de poder hacer sugerencias para una mayor mejora. Este proceso implicaría el analizar los problemas ya en la etapa de recolección de datos, problemas en la implementación de las directivas de calidad y problemas en la interpretación de los datos.

Basili [BAS96] ha proporcionado una serie de plantillas que son útiles para los desarrolládores que deseen utilizar el método OPM para desarrollar métricas realistas sobre sus proyectos. Los objetivos de OPM pueden articularse por medio de tres plantillas que cubren el propósito, la perspectiva y el entorno.

La plantilla o esquema de cálculo denominada de propósito se utiliza para articular o comparar lo que está siendo analizado y el propósito de dicha parte del proyecto. Por ejemplo, un diseñador puede desear analizar la efectividad de las revisiones de diseño con el propósito de mejorar la tasa de eliminación de errores de dicho proceso o el propio diseñador puede desear analizar las normas utilizadas por su empresa con el objetivo de reducir la cantidad de mano de obra utilizada durante el mantenimiento. Una segunda plantilla está relacionada con la perspectiva. Esta plantilla pone su atención en los factores que son importantes dentro del propio proceso o producto que está siendo evaluado. Ejemplos típicos de esto incluyen los factores de calidad de la mantenibilidad, chequeo, uso y otros factores tales como el coste y la corrección. Esta plantilla se fundamenta en la perspectiva del propio proceso al que se dirige.

Hay varios enfoques que pueden hacerse sobre el proceso de desarrollo de software - e l del cliente y el del diseñador son los dos más típicos — y la elección de una u otra perspectiva tiene un efecto muy grande sobre los análisis que se llevan a cabo. **Por** ejemplo, si un cierto proceso como una revisión de requisitos está siendo analizada con respecto al coste, la perspectiva del diseñador, es la del que desea reducir el coste del proceso en términos de hacerlo más eficiente en cuanto a la detección de errores; sin embargo, si después examinamos el mismo propósito pero desde el punto de vista del cliente, concretamente sobre si su personal puede emplear o no una gran cantidad de tiempo en dichas revisiones, entonces la perspectiva podría involucrar el plantearse un uso más eficiente de dicho tiempo empleado en las revisiones. Ambas perspectivas son válidas —a veces se solapan y a veces son antitéticas — existe, por ejemplo, una necesidad natural en el diseñador de maximizar el beneficio a la vez que el objetivo del cliente es la corrección máxima del producto y la entrega dentro del plazo. Un punto importante a recalcar es que del examen de la perspectiva del producto, el desarrollador está en una mejor posición para evaluar un proceso de software y un producto de software y también la mejora de los mismos.

Una tercera plantilla implica el entorno. Este es el contexto dentro del cual el método OPM se aplica e implica el examen del personal, la propia empresa y los entornos de recursos en los que el análisis se está llevando a cabo. Factores típicos de entorno incluyen, por ejemplo, el tipo de sistema informático que está siendo usado, las habilidades del personal implicado en el proyecto, el modelo de ciclo de vida adoptado para tal caso, la cantidad de recursos adiestrados disponibles y el área de aplicación.

Una vez que tanto el propósito como la perspectiva y el entorno de un objetivo han sido bien especificados, el proceso de planteamiento de cuestiones y el desarrollo de una métrica o valoración puede comenzar. Antes de examinar este proceso, merece la pena dar algunos ejemplos de objetivos empleados en los términos planteados. El primero se describe a continuación:

> El objetivo es analizar los medios por **los** que revisamos el código de programación con el propósito de evaluar la efectividad en la detección de errores desde el punto de vista del gestor del proyecto de software dentro de los proyec**tos** que suministran programas críticos bajo el punto de vista de la seguridad.

En el ejemplo planteado, el propósito es la evaluación, la perspectiva es la eliminación de defectos bajo el punto de vista del gestor de proyectos dentro del entorno de aplicaciones en los que la seguridad es un aspecto crítico. Otro ejemplo es:

> Nosotros deseamos examinar la efectividad de las normas de programación que usamos para el lenguaje de programación C++, con el fin de determinar si son efectivos en la producción de software, que pueda ser reutilizado dentro de otros proyectos. En particular, estamos interesados en lo que se necesita con el fin de organizar efectivamente un departamento nuevo encargado de el mantenimiento de una biblioteca de software reutilizable. En este estudio se da el software de nuestra área de aplicación principal; que es la de control de inventarios.

Aquí, el propósito es la mejora de un estándar de programación; la perspectiva es la de la reutilización bajo el punto de vista del personal encargado de la administración de una biblioteca de software reutilizable, y el entorno son todos aquellos proyectos que impliquen funciones de control de inventarios.

Otro ejemplo adicional es el siguiente:

> Deseamos examinar el efecto de utilizar un constructor de interfaces gráficas, sobre la mejora de las interfaces hombre-máquina, que producimos para nuestros sistemas de administración. En particular, deseamos examinar cómo puede esto afectar a la facilidad de manejo de estas interfaces por parte del usuario. Un foco de atención principai es cómo percibirán **los** usuarios de estos sistemas que dichas interfaces han cambiado.

Aquí, el propósito es analizar una herramienta-con el objetivo de determinar una mejora con respecto a la facilidad de uso, bajo el punto de vista del usuario dentro del contexto de los sistemas administrativos. Un ejemplo final es el siguiente:

> Nuestro objetivo **es** examinar el proceso de chequeo de módulos de código de forma que podamos utilizar los resultados de las comprobaciones con el fin de predecir el número de defectos de código en comprobaciones futuras. La perspectiva que deseamos tener surge de una preocupación sobre el exceso de errores que se han cometido y que no han sido detectados hasta el chequeo del sistema. Deseamos ver qué factores son importantes que permitan que un programador tome decisiones sobre si un módulo está disponible para ser entregado a los probadores del sistema, o por el contrario requiere una comprobación adicional. Deseamos concentramos en nuestro modelo de ciclo de vida actual con programadores que utilizan el lenguaje de programación FORTRAN.

Aquí, el objetivo es la predicción, el análisis de un proceso —el de chequeo de código — la perspectiva es la de reducción de costes a través de una reducción del número de errores residuales, que se deslizan dentro del propio chequeo del sistema. La perspectiva es desde el punto de vista del programador y el entorno el de los proyectos convencionales que utilizan el lenguaje de programación FORTRAN.

La plantilla propósito se utiliza para definir lo que está siendo estudiado: podría ser un proceso, un producto, un estándar o un procedimiento. La plantilla también define lo que se va a hacer, y la razón de hacerlo. La perspectiva tiene el objetivo de asegurar que los objetivos no son demasiado ambiciosos: al final del día el método OPM requerirá la realización de medidas y estas medidas y los datos estadísticos asociados serán sólo efectivos cuanto más grande sea el número de factores dentro de un nivel razonable. La plantilla del entorno tiene una función similar: reduce el factor de espacio y permite que sean hechas comparaciones estadísticas efectivas entre procesos y productos.

Con el fin de terminar esta sección es provechoso analizar el método OPM en acción sobre un pequeño ejemplo, con el siguiente objetivo de proceso:

Analícese el proceso de confección de prototipos con el propósito de evaluar desde el punto de vista del desarrollador, el número de requisitos que se rechazan **por** el cliente en una última etapa del proyecto.

Nosotros asumiremos un modelo desechable de confección de prototipos en el que se use una tecnología como la de un lenguaje típico de cuarta generación para desarrollar una versión rápida de un sistema que, cuando el cliente lo ha aceptado, se implemente de forma convencional con la eliminación del propio prototipo. Esto plantea un cierto número de preguntas que entonces pueden ser procesadas con el fin de evaluar el proceso de confección de prototipos desechables:

- ¿Qué medida tomaríamos con los requerimientos que se hayan cambiado durante la parte convencional del proyecto?
- ¿Se deben ponderar todos los requisitos de forma igualitaria o son algunos más complejos que otros?
- ¿Qué tipos de cambios en los requisitos debemos considerar? Después de todo, algunos de ellos serán debidos a imperfecciones en el proceso de confección de prototipos mientras que otros podrían tener que ver con factores extraños que pueden ser debidos a cambios en las propias circunstancias del cliente.

Con el fin de aplicar el método OPM, necesitamos **un** modelo del proceso que esté llevándose a cabo y un modelo de la perspectiva de calidad: el número de cambios en los requisitos que son exigidos por el cliente no provienen generalmente de cambios en sus propias circunstancias.

Supongamos que el modelo para el proceso de confección de prototipos desechables es:

1. Especificar los requisitos.
2. Valorar cada requisito en importancia según términos del cliente.
3. Valorar cada requisito en términos de la complejidad de su descripción.
4. Planificar una serie de reuniones de revisión en las que se presente a través de un prototipo una cierta selección de requisitos donde el gestor del proyecto tome una decisión sobre en qué se basa la selección de una presentación de, por ejemplo, dos horas de duración.

Supongamos un modelo muy simple de perspectiva de calidad, dónde cada requisito R_i esté asociado con una complejidad ponderada C_i. Entonces el tamaño de los requisitos totales será:

$$\sum_{i=0}^{n} R_i \cdot C_i$$

Supongamos que una cierta proporción p de estos requisitos han sido puestos en cuestión emprendiéndose un chequeo de aceptación por parte del cliente, y que estos requisitos no han sido debidos a cambios en las propias circunstancias del cliente. Entonces, la métrica

$$e = p \cdot \sum_{i=0}^{n} R_i \cdot C_i$$

representa una cierta medida de la desviación de los requisitos desde el prototipo a lo largo del chequeo de aceptación.

Este valor puede entonces ser comparado con los valores base de otros proyectos de confección de prototipos que tengan un entorno parecido. Si se ha observado una cierta mejora, la siguiente etapa es intentar descubrir cómo se ha logrado esta mejora, por ejemplo, en este proyecto el cliente puede haber enviado el mismo representante para ayudar en las demostraciones del prototipo y por consiguiente ha conseguido una consistencia mayor que faltaba en otros proyectos, o el personal del desarrollo que estaba implicado en el proyecto puede haber estado formado por analistas en lugar de diseñadores, y así haber constituido una más sólida relación con el cliente. Si se observaba un incremento en la métrica e, entonces un análisis similar debería llevarse a cabo en términos de lo que constituían los factores desfavorables que afectaban al proyecto.

Lo ya expuesto, entonces, ha sido un resumen esquemático del proceso OPM. Un punto importante a considerar es que ya que existe un número casi infinito de métricas que pueden usarse para caracterizar un producto de software y un proceso de software existe una necesidad de determinar la forma de seleccionarlas, o dicho de otra forma, cuál de las OPM es el mejor ejemplo. Una vez que esto ha sido tomado en consideración en una empresa, esa empresa puede entonces implicarse en una actividad continua de mejora de procesos, que la coloque en un nivel 4 ó 5 de la Escala de Modelos de Madurez de Capacidades y así distinguirla de aquellas otras que lleguen sólo a niveles 1 ó 2.

4.8 VARIACIÓN DE LA GESTIÓN: CONTROL DE PROCESOS ESTADÍSTICOS

Debido a que el proceso de software y el producto que tal proceso produce son ambos influenciados por muchos parámetros (por ejemplo: el nivel de habilidad de los realizadores de dichos procesos, la estructura del equipo de software, el conocimiento del cliente, la tecnología que va a ser implementada, las herramientas que serán usadas en la actividad de desarrollo), la métrica elegida para un proyecto o producto no será la misma que otras métricas similares seleccionadas para otro proyecto. En efecto, hay a menudo variaciones significativas en las métricas elegidas como parte del proceso de software.

Puesto que la misma métrica de procesos variará de un proyecto a otro proyecto, ¿cómo podemos decir si unos valores de métricas mejoradas (o degradadas) que ocurren como consecuencia de actividades de mejora están

de hecho teniendo un impacto cuantitativo real? ¿Cómo saber si lo que nosotros estamos contemplando es una tendencia estadísticamente válida o si esta «tendencia» es simplemente el resultado de un ruido estadístico? ¿Cuándo son significativos los cambios (ya sean positivos o negativos) de una métrica de software particular?

Cita: ...reducir mi mensaje para la gestión ... pocas palabras, diría todo lo que hoy ... hacer con reducir la variación.
W. Edwards Deming

Se dispone de una técnica gráfica para determinar si los cambios y la variación en los datos de la métrica son significativos. Esta técnica llamada *gráfico **de** control* y desarrollada por Walter Shewart en 1920[10], permite que los individuos o las personas interesadas en la mejora de procesos de software determine si la dispersión (variabilidad) y «la localización» (media móvil) o métrica de procesos que es *estable* (esto es, si el proceso exhibe cambios controlados o simplemente naturales) o *inestable* (esto es, si el proceso exhibe cambios fuera de control y las métricas no pueden usarse para predecir el rendimiento). Dos tipos diferentes de gráficos de control se usan en la evaluación de los datos métricos [ZUL99]: (1) el gráfico de control de rango móvil (Rm) y (2) el gráfico de control individual.

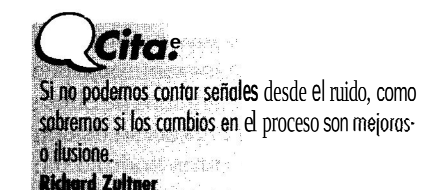
Para ilustrar el enfoque que significa un gráfico de control, consideremos una organización de software que registre en la métrica del proceso los errores descubiertos por hora de revisión, E_r. Durante los pasados 15 meses, la organización ha registrado el E_r para 20 pequeños proyectos en el mismo dominio de desarrollo de software general. Los valores resultantes para E_r están representados en la figura 4.8. Si nos referimos a la figura, E_r varía desde una tasa baja de 1.2 para el proyecto *3* a una tasa más alta de 5.9 para el proyecto 17. En un esfuerzo de mejorar la efectividad de las revisiones, la organización de software proporcionaba entrenamiento y asesoramiento a todos los miembros del equipo del proyecto, comenzando con el proyecto 11.

¿Cómo podemos estar seguros de que las métricas que hemos recopilado son estadísticamente validas?

Richard Zultner proporciona una vista general del procedimiento que se requiere para desarrollar un *gráfico de control **de** rango móvil* (Rm) para determinar la estabilidad del proceso [ZUL99]:

1. Calcular los rangos móviles: el valor absoluto de las diferencias sucesivas entre cada pareja de puntos de datos... Dibujar estos rangos móviles sobre el gráfico.
2. Calcular la media de los rangos móviles... dibujando ésta («barra Rm») como la línea central del propio gráfico.
3. Multiplicar la media por 3.268. Dibujar esta línea como el *límite de control superior* [LCS]. Esta línea supone tres veces el valor de la desviación estándar por encima de la media.

Usando los datos representados en la figura **4.8** y los distintos pasos sugeridos por Zultner como anteriormente se ha descrito, desarrollamos un gráfico de control Rm que se muestra en la Figura 4.9. El valor (medio) «barra Rm» para los datos de rango móvil es 1.71. El límite de control superior (LCS) es 5.57.



Para determinar si la dispersión de las métricas del proceso es estable puede preguntarse una cuestión muy sencilla: ¿Están los valores de rango móvil dentro del LCS? Para el ejemplo descrito anteriormente, la contestación es «sí». Por consiguiente, la dispersión de la métrica es estable.

El gráfico de control individual se desarrolla de la manera siguiente[11]:

1. Dibujar los valores de la métrica individual según *se* describe en la Figura 4.8.
2. Calcular el valor promedio, A_m, para los valores de la métrica.
3. Multiplicar la media de los valores Rm (la barra Rm) por 2.660 y añadir el valor de A_m calculado en el paso 2. Esto da lugar a lo que se denomina *límite **de** proceso natural superior (LPNS)*. Dibujar el LPNS.
4. Multiplicar la media de los valores Rm (la barra Rm) por 2.660 y restar este valor del A_m calculado en el paso 2. Este cálculo da lugar al *límite de proceso natural inferior (LPNZ)*. Dibujar el LPNI. Si el LPNI es menor que 0.0, no necesita ser dibujado a menos que la métrica que está siendo evaluada tome valores que sean menores que 0.0.
5. Calcular la desviación estándar según la fórmula (LPNS – A_m)/3. Dibujar las líneas de la desviación estándar una y dos por encima y por debajo de A_m. Si cualquiera de las líneas de desviación estándar es menor de 0.0, no necesita ser dibujada a menos que la métrica que está siendo evaluada tome valores que sean menores que 0.0.

[10] Debería tenerse en cuenta que aunque el gráfico de control fue desarrollado originalmente para procesos de fabricación es igualmente aplicable a procesos de software.

[11] El estudio que sigue *es* un resumen de los pasos sugeridos por Zultner [ZUL99].

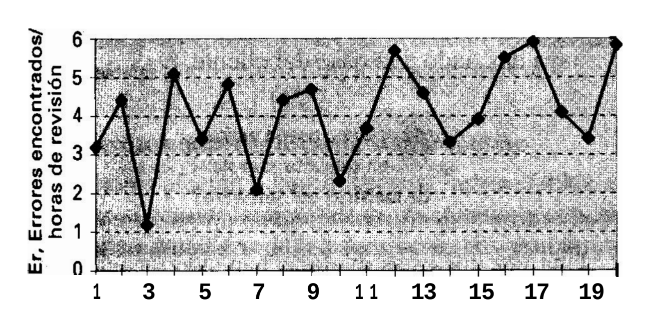


FIGURA 4.8. Datos de métricas para descubrir errores por hora de revisión.

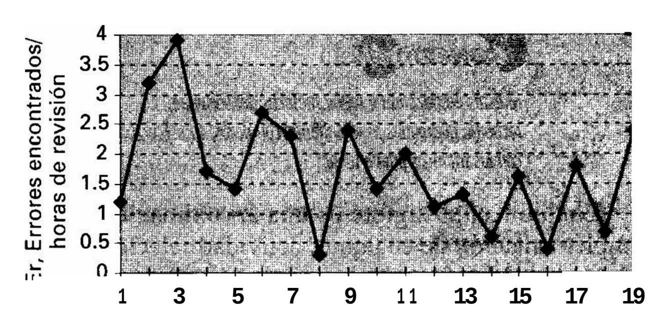


FIGURA 4.9. Gráfico de control de rango móvil (Rm).

Aplicando estos pasos a los datos representados en la Figura 4.8, se llega a un *gráfico de control individual* según se ve en la figura 4.10.



Se puede encontrar un Gráfico de Control Común que cubre el tema en alguna dimensión en:
www.sytsma.com/tqmtools/ctlchfprinciples.html

Zultner [ZUL99] revisa cuatro criterios, denominados *reglas de zona*, que pueden usarse para evaluar si los cambios representados por la métrica indican que un proceso está bajo control o fuera de control. Si cualquiera de las siguientes condiciones es verdadera, los datos de la métrica indican un proceso que está fuera de control:

1. Un valor de la métrica individual aparece fuera del **LPNS.**
2. Dos de cada tres valores de métricas sucesivas aparecen más de dos desviaciones estándar fuera del valor A,.
3. Cuatro de cada cinco valores de métricas sucesivas aparecen alejados más de una desviación estándar del valor *A,.*
4. Ocho valores consecutivos de métrica aparecen todos situados a un lado del valor *A,.*

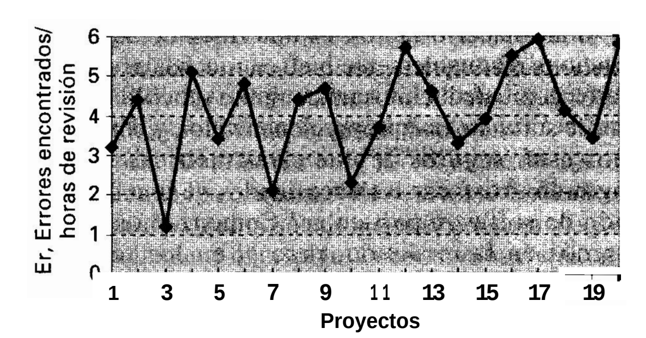


FIGURA 4.10. Gráfico de control individual.

Puesto que todas estas condiciones fallan para los valores mostrados para la figura 4.10, se concluye que los datos de las métricas se derivan de un proceso estable y que se pueden deducir legítimamente a partir de los datos recogidos en la métrica una información que constituye una verdadera tendencia. Si nos referimos a la figura 4.10, puede verse que la variabilidad de E_r decrece a partir del proyecto 10 (esto es, después de un esfuerzo para mejorar la efectividad de las revisiones). Calculando el valor medio para los 10 primeros y los 10 últimos proyectos, puede demostrarse que el valor medio de E_r para los proyectos del 11 al 20, muestran un 29 por 100 de mejora en relación con la E_r de los proyectos 1 al 10. Puesto que el gráfico de control indica que el proceso es estable, parece que los esfuerzos para mejorar la efectividad de la revisión dan sus resultados.

4.9 MÉTRICA PARA ORGANIZACIONES PEQUEÑAS

La amplia mayoría de las organizaciones de desarrollo de software tienen menos de 20 personas dedicadas al software. Es poco razonable, y en la mayoría de los casos no es realista, esperar que organizaciones como éstas desarrollen programas métricos de software extensos. Sin embargo, si que es razonable sugerir que organizaciones de software de todos los tamaños midan y después utilicen las métricas resultantes para ayudar a mejorar sus procesos de software local y la calidad y oportunidad de los productos que realizan. Kautz [KAU99] describe un escenario típico que ocurre cuando se piensa en programas métricos para organizaciones pequeñas de software:

> Originalmente, los desarrolladores de software acogían nuestras actividades con un alto grado de escepticismo, pero **al** final **las** aceptaban debido **a** que nosotros conseguíamos que nuestras medidas fueran simples de realizar, adaptadas a cada organización y se aseguraba que dichas medidas producían una información válida y útil. **Al** final, los programas proporcionaban una base para encargarse de **los** clientes y para **la** planificación y desarrollo de su trabajo futuro.

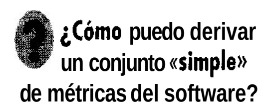


*Si estás empezando a reunir métricas del **software**, recuerda guardarlas. Si te entierras con datas, tu esfuerzo con los métricos puede fallar.*

Lo que Kautz sugiere es una aproximación para la implementación de cualquier actividad relacionada con el proceso del software: que sea simple; adaptada a satisfacer las necesidades locales y que se constate que realmente añada valor. En los párrafos siguientes examinamos cómo se relacionan estas líneas generales con las métricas utilizadas para negocios pequeños.

«Para hacerlo fácil», es una línea de acción que funciona razonablemente bien en muchas actividades. Pero ¿cómo deducimos un conjunto sencillo de métricas de software que proporcionen valor, y cómo podemos estar seguros de que estas métricas sencillas lograran satisfacer las necesidades de una organización de software particular? Comenzamos sin centrarnos en la medición, pero **sí** en los resultados finales. El grupo de software es interrogado para que defina un objetivo simple que requiera mejora. Por ejemplo, «reducir el tiempo para evaluar e implementar las peticiones de cambio». Una organización pequeña puede seleccionar el siguiente conjunto de medidas fácilmente recolectables:

- Tiempo (horas o días) que transcurren desde el momento que es realizada una petición hasta que se complete su evaluación, t_{cola}.
- Esfuerzo (horas-persona) para desarrollar la evaluación, W_{eval}.
- Tiempo (horas o días) transcurridos desde la terminación de la evaluación a la asignación de una orden de cambio al personal, t_{eval}.
- Esfuerzo (horas-persona) requeridas para realizar el cambio, W_{cambio}.
- Tiempo requerido (horas o días) para realizar el cambio, t_{cambio}.
- Errores descubiertos durante el trabajo para realizar el cambio, E_{cambio}.
- Defectos descubiertos después de que el cambio se haya desviado **a** la base del cliente, D_{cambio}.



¿Cómo puedo derivar un conjunto «simple» de métricas del software?

Una vez que estas medidas han sido recogidas para un cierto número de peticiones de cambio, es posible calcular el tiempo total transcurrido desde la petición de cambio hasta la implementación de dicho cambio y el porcentaje de tiempo consumido por el proceso de colas iniciales, la asignación de cambio y evaluación, y la implementación del cambio propiamente dicho. De forma similar, el porcentaje de esfuerzo requerido para la evaluación y la implementación puede también ser determinado. Estas métricas pueden ser evaluadas en el contexto de los datos de calidad, E_{cambio} y D_{cambio}. Los porcentajes proporcionan un análisis interno para buscar el lugar donde los procesos de petición de cambio se ralentizan y pueden conducir a unos pasos de mejoras de proceso para reducir t_{cola}, W_{eval}, t_{eval}, W_{cambio}, y/o E_{cambio}. Además, la eficiencia de eliminación de defectos (EED) puede ser calculada de la siguiente manera:

$$EED = E_{cambio} / (E_{cambio} + D_{cambio})$$

EED puede compararse con el tiempo transcurrido y el esfuerzo total para determinar el impacto de las actividades de aseguramiento de la calidad sobre el tiempo y el esfuerzo requeridos para realizar un cambio.

Para grupos pequeños, el coste de incorporar medidas y métricas de cálculo oscila entre el *3* y el **8** por 100 del presupuesto del proyecto durante la fase de aprendizaje y después cae a menos del 1 por 100 del presupuesto del proyecto una vez que la ingeniería del software y la gestión de proyectos se hayan familiarizado con el programa de métricas [GRA99]. Estos costes pueden representar una mejora de las inversiones siempre que el análisis derivado a partir de los datos de la métrica conduzcan a una mejora de procesos significativa para la organización del software.

4.10 ESTABLECIMIENTO DE UN PROGRAMA DE MÉTRICAS DE SOFTWARE

El Instituto de Ingeniería del Software (IIS) ha desarrollado una guía extensa [PAR96] para establecer un programa de medición de software dirigido hacia objetivos. La guía sugiere los siguientes pasos para trabajar:

1. Identificar los objetivos del negocio.
2. Identificar lo que se desea saber o aprender.
3. Identificar los subobjetivos.
4. Identificar las entidades y atributos relativos a esos subobjetivos.
5. Formalizar los objetivos de la medición.
6. Identificar preguntas que puedan cuantificarse y los indicadores relacionados que se van a usar para ayudar a conseguir los objetivos de medición.
7. Identificar los elementos de datos que se van a recoger para construir los indicadores que ayuden a responder a las preguntas planteadas.
8. Definir las medidas a usar y hacer que estas definiciones sean operativas.
9. Identificar las acciones que serán tomadas para mejorar las medidas indicadas.
10. Preparar un plan para implementar estas medidas.



Se puede descargar una guía para la medición del software orientado a objetivos desde: **www.sei.tmu.edu**

Si se quiere entrar en una discusión más profunda de los pasos anteriores, lo mejor es acudir directamente a la guía ya comentada del IIS (SEI) Instituto de Ingeniería del Softwate. Sin embargo, merece la pena repasar brevemente los puntos clave de ésta.

Ya que el software, en primer lugar, soporta las funciones del negocio, en segundo lugar, diferencia o clasifica los sistemas o productos basados en computadora, y en tercer lugar puede actuar como un producto en sí mismo, los objetivos definidos para el propio negocio pueden casi siempre ser seguidos de arriba abajo hasta los objetivos más específicos a nivel de ingeniería de software. Por ejemplo, considérese una compañía que fabrica sistemas avanzados para la seguridad del hogar que tienen un contenido de software sustancial. Trabajando como un equipo, los ingenieros de software y los gestores del negocio, pueden desarrollar una lista de objetivos del propio negocio convenientemente priorizados:

1. Mejorar la satisfacción de nuestro cliente con nuestros productos.
2. Hacer que nuestros productos sean más fáciles de usar.
3. Reducir el tiempo que nos lleva sacar un nuevo producto al mercado.
4. Hacer que el soporte que se dé a nuestros productos sea más fácil.
5. Mejorar nuestro beneficio global.



CLAVE

las métricas del software que elijas estarbn tonducidas por el negocio o por los objetivos técnicos que desees cumplir.

La organización de software examina cada objetivo de negocios y se pregunta: «¿Qué actividades gestionaremos (ejecutaremos) y qué queremos mejorar con estas actividades?» Para responder a estas preguntas el IIS recomienda la creación de una «lista de preguntas-entidad», en la que todas las cosas (entidades) dentro del proceso de software que sean gestionadas o estén influenciadas por la organización de software sean anotadas convenientemente. Ejemplo de tales entidades incluye: recursos de desarrollo, productos de trabajo, código fuente, casos de prueba, peticiones de cambio, tareas de ingeniería de software y planificaciones. Para cada entidad listada, el personal dedicado al software desarrolla una serie de preguntas que evalúan las características cuantitativas de la entidad (por ejemplo: tamaño, coste, tiempo para desarrollarlo). Las cuestiones derivadas como una consecuencia de la creación de una lista tal de preguntas-entidad, conducen a la derivación de un conjunto de objetivos de segundo nivel (subobjetivos) que se relacionan directamente con las entidades creadas y las actividades desarrolladas como una parte del proceso del software.

Considérese el cuarto objetivo apuntado antes: «Hacer que el soporte para nuestros productos sea más fácil». La siguiente lista de preguntas pueden ser derivadas a partir de este objetivo [PAR96]:

- ¿Contienen las preguntas de cambio del cliente la información que nosotros requerimos para evaluar adecuadamente el cambio y de esa forma realizarle en un tiempo y formas oportunos?
- ¿Cómo es de grande el registro de peticiones de cambio?
- ¿Es aceptable nuestro tiempo de respuesta para localizar errores de acuerdo a las necesidades del cliente?
- ¿Se sigue convenientemente nuestro proceso de control de cambios (Capítulo 9)?
- ¿Se llevan a cabo los cambios de alta prioridad de manera oportuna y sincronizada?

Basándose en estas cuestiones, la organización de software puede deducir los objetivos de segundo nivel (subobjetivos) siguientes: Mejorar el rendimiento del proceso de gestión del cambio. Las entidades de proceso de software y atributos que son relevantes para los propósitos u objetivos más específicos o de segundo nivel son identificados y se definen además las metas u objetivos de medida asociados con dichos atributos.

El IIS [PAR96] proporciona una guía detallada para los pasos 6 al 10 de este enfoque de medida orientado hacia objetivos. En esencia, se aplica un proceso de refinamiento por pasos en el que los objetivos son refinados en preguntas que son a su vez refinadas de forma más detallada en entidades y atributos que por Último se analizan en un Último paso de forma más minuciosa a nivel de la métrica en **sí.**

RESUMEN

La medición permite que gestores y desarrolladores mejoren el proceso del software, ayuden en la planificación, seguimiento y control de un proyecto de software, y evalúen la calidad del producto (software) que se produce. Las medidas de los atributos específicos del proceso, del proyecto y del producto se utilizan para calcular las métricas del software. Estas métricas se pueden analizar para proporcionar indicadores que guían acciones de gestión y técnicas.

Las métricas del proceso permiten que una organización tome una visión estratégica proporcionan-

do mayor profundidad de la efectividad de un proceso de software. Las métricas del proyecto son tácticas. Estas permiten que un gestor de proyectos adapte el enfoque a flujos de trabajo del proyecto y a proyectos técnicos en tiempo real.

Las métricas orientadas tanto al tamaño como a la función se utilizan en toda la industria. Las métricas orientadas al tamaño hacen uso de la línea de código como factor de normalización para otras medidas, como persona-mes o defectos. El punto de función proviene de las medidas del dominio de información y de una evaluación subjetiva de la complejidad del problema.

Las métricas de la calidad del software, como métricas de productividad, se centran en el proceso, en el proyecto y en el producto. Desarrollando y analizando una línea base de métricas de calidad, una organización puede actuar con objeto de corregir esas áreas de proceso del software que son la causa de los defectos del software.

Las métricas tienen significado solo **si** han sido examinadas para una validez estadística. El gráfico de control es un método sencillo para realizar esto y al mismo tiempo examinar la variación y la localización de los resultados de las métricas.

La medición produce cambios culturales. La recopilación de datos, el cálculo de métricas y la evaluación de métricas son los tres pasos que deben implementarse al comenzar un programa de métricas. En general, un enfoque orientado a los objetivos ayuda a una organización a centrarse en las métricas adecuadas para su negocio. Los ingenieros del software y sus gestores pueden obtener una visión más profunda del trabajo que realizan y del producto que elaboran creando una línea base de métricas —una base de datos que contenga mediciones del proceso y del producto–.

REFERENCIAS

[ALA97] Alain, A., M. Maya, J. M. Desharnais y S. St. Pierre, «Adapting Function Points to Real-Time Software», *American Programmer,* vol. 10, n.º 11, Noviembre 1997, pp. 32-43.

[ART85] Arthur, L. J., *Measuring Programmer Productivity and Software Quality,* Wiley-Interscience, 1985.

[ALB79] Albrecht, A. J., «Measuring Application Development Productivity», *Proc. IBM Application Development Symposium,* Monterey, CA, Octubre 1979, pp. 83-92.

[ALB83] Albretch, A. J., y J. E. Gaffney, «Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation», *IEEE Trans. Software Engineering,* Noviembre 1983, pp. 639-648.

[BOE81] Boehm, B., *Software Engineering Economics,* Prentice Hall, 1981.

[GRA87] Grady, R.B., y D.L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, 1987.

[GRA92] Grady, R.G., *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, 1992.

[GRA94] Grady, R., «Succesfully Applying Software Metrics», *Computer,* vol. 27, n.° 9, Septiembre 1994, pp. 18-25.

[GRA99] Grable, R., et al., «Metrics for Small Projects: Experiences at SED», *IEEE Software,* Marzo 1999, pp. 21-29.

[GIL88] Gilb, T., *Principles of Software Project Management,* Addison-Wesley, 1998.

[HET93] Hetzel, W., *Making Software Measurement Work,* QED Publishing Group, 1993.

[HUM95] Humphrey, W., *A Disciplinefor Software Engineering*, Addison-Wesley, 1995.

[IEE93] *IEEE Software Engineering Standards,* Std. 610.12-1990, pp. 47-48.

[IFP94] *Function Point Counting Practices Manual,* Release 4.0, InternationalFunction Point Users Group (IFPUG), 1994.

[JON86] Jones, C., *Programming Productivity,* McGraw-Hill, 1986.

[JON91] Jones, C., *Applied Software Costs,* McGraw-Hill, 1991.

[JON98] Jones, C., *Estimating Software Costs,* McGraw-Hill, 1998.

[KAU99] Kautz, K., «Making Sense of Measurement for Small Organizations», *IEEE Software,* Marzo 1999, pp. 14-20.

[MCC78] McCall, J. A., y J. P. Cavano, «A Framework for the Measurement of Software Quality», *ACM Software Quality Assurance Workshop,* Noviembre 1978.

[PAU94] Paulish, D., y A. Carleton, «Case Studies of Software Process Improvement Measurement», *Computer,* vol. 27, n.° 9, Septiembre 1994, pp. 50-57.

[PAR96] Park, R. E., W. B. Goethert y W. A. Florac, *Goal Driven Software Measurement-A Guidebook*, CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, Agosto 1996.

[RAG95] Ragland, B., «Measure, Metnc or Indicator: What'c the Difference?», *Crosstalk,* vol. 8, n.° 3, Marzo 1995, pp. 29-30.

[TAL81] Tjima, D., y T. Matsubara, «The Computer Software Industry in Japan», *Computei;* Mayo 1981, p. 96.

[WHI95] Whitmire, S. A., «An Introduction to 3D Function Points», *Software Development,* Abril 1995, pp. 43-53.

[ZUL99] Zultner, R. E., «What Do Our Metrics Mean?», *Cutter IT Journal,* vol. 12, n.° 4, Abril 1999, pp. 11-19.

PROBLEMAS Y PUNTOS A CONSIDERAR

4.1. Sugiera tres medidas, tres métricas y los indicadores que se podrían utilizar para evaluar un automóvil.

4.2. Sugiera tres medidas, tres métricas y los indicadores correspondientes que se podrían utilizar para evaluar el departamento de servicios de un concesionario de automóviles.

4.3. Describa, con **sus** propias palabras, la diferencia entre métricas del proceso y del proyecto.

4.4. ¿**Por** qué las métricas del software deberían mantenerse «privadas»? Proporcione ejemplos de tres métricas que debieran ser privadas. Proporcione ejemplos de tres métricas que debieran ser públicas.

4.5. Obtenga una copia de [HUM95] y escriba un resumen en una o dos páginas que esquematice el enfoque PSP.

4.6. Grady sugiere una etiqueta para las métricas del software. ¿Puede añadir más reglas a las señaladas en la Sección 4.2.1?

4.7. Intente completar el diagrama en espina de la Figura 4.3. Esto es, siguiendo el enfoque utilizado para especificaciones «incorrectas», proporcione información análoga para «perdido, ambiguo y cambios».

4.8. ¿**Qué** es una medida indirecta y por qué son comunes tales cambios en un trabajo de métricas de software?

4.9. **El** equipo A encontró 342 errores durante el proceso de ingeniería del software antes de entregarlo. El equipo **B** encontró 184 errores. ¿Qué medidas adicionales se tendrían que tomar para que los proyectos A y **B** determinen qué equipos eliminaron los errores más eficientemente? ¿Qué métricas propondrían para ayudar a tomar determinaciones? ¿Qué datos históricos podrían ser útiles?

4.10. Presente un argumento en contra de las líneas de código como una medida de la productividad del software. ¿Se va a sostener **su** propuesta cuando se consideren docenas ***o*** cientos de proyectos?

4.11. Calcule el valor del punto de función de un proyecto con las siguientes características del dominio de información:

Número de entradas de usuario: 32

Número de salidas de usuario: 60

Número de peticiones de usuario: 24

Número de archivos: 8

Número de interfaces externos: 2

Asuma que todos los valores de ajuste de complejidad están en la media.

4.12. Calcule el valor del punto de función de un sistema empotrado con las características siguientes:

Estructuras de datos interna: 6
Estructuras de datos externa: 3
Número de entradas de usuario: 12
Número de salidas de usuario: 60
Número de peticiones de usuario: 9
Número de interfaces externos: 3
Transformaciones: 36
Transiciones: 24

Asuma que la complejidad de las cuentas anteriores se divide de igual manera entre bajo, medio y alto.

4.13. El software utilizado para controlar una fotocopiadora avanzada requiere 32.000 líneas de C y 4.200 líneas de Smalltalk. Estime el número de puntos de función del software de la fotocopiadora.

4.14. McCall y Cavano (Sección 4.5.1) definen un «marco de trabajo» de la calidad del software. Con la utilización de la información de este libro y de otros se amplían los tres «puntos de vista» importantes dentro del conjunto de factores y de métricas de calidad.

4.15. Desarrolle sus propias métricas (*no* utilice las presentadas en este capítulo) de corrección, facilidad de mantenimiento, integridad y facilidad de uso. Asegúrese de que se pueden traducir en valores cuantitativos.

4.16. ¿Es posible que los desperdicios aumenten mientras que disminuyen defectos/KLDC? Explíquelo.

4.17. ¿Tiene algún sentido la medida LDC cuando se utiliza el lenguaje de cuarta generación? Explíquelo.

4.18. Una organización de software tiene datos EED para 15 proyectos durante los 2 últimos años. Los valores recogidos son: 0.81, 0.71, 0.87, 0.54, 0.63, 0.71, 0.90, 0.82, 0.61, 0.84, 0.73, 0.88, 0.74, 0.86, 0.83. Cree Rm y cuadros de control individuales para determinar si estos datos se pueden utilizar para evaluar tendencias.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

La mejora del proceso del software (MPS) ha recibido una significativa atención durante la pasada década. Puesto que la medición y las métricas del software son claves para conseguir una mejora del proceso del software, muchos libros sobre MPS también tratan las métricas. Otras lecturas adicionales que merecen la pena incluyen:

Burr, A., y M. Owen, *Statistical Methosfor Software Quality,* Intemational Thomson Publishing, 1996.

Florac, W. A., y A. D. Carleton, *Measuring the Software Process: Statistical Process Control* ***for*** *Software Process lmprovement,* Addison-Wesley, 1999.

Garmus, D., y D. Herron, *Measuring the Software Process: A Practica1 Guide to Functional Measurements,* Prentice-Hall, 1996.

Kan, S. H., *Metrics and Models in Software Quality Engineering,* Addison-Wesley, 1995.

Humphrey [HUM95], Yeh *(Software Process Control,* McGraw-Hill, 1993), Hetzel [HET93] y Grady [GRA92] estudian cómo **se** pueden utilizar las métricas del software para proporcionar los indicadores necesarios que mejoren el proceso del software. Putnam y Myers (*Executive Briefing: Controlling Software Development,* IEEE Computer Society, 1996) y Pul-

ford y sus colegas (*A Quantitative Approach to Software Management*, Addison-Wesley, 1996) estudian las métricas del proceso y su uso desde el punto de vista de la gestión.

Weinberg (*Quality Software Management, Volume 2: First Order Measurement*, Dorset House, 1993) presenta un modelo útil para observar proyectos de software, asegurándose el significado de la observación y determinando el significado de las decisiones tácticas y estratégicas. Garmus y Herron (*Measuring the Software Process*, Prentice-Hall, 1996) trata las métricas del proceso en el análisis del punto de función. El Software Productivity Consortium (*The Software Measurement Guidebook*, Thomson Computer Press, 1995) proporciona sugerencias útiles para instituir un enfoque efectivo de métricas. Oman y Pfleeger (*Applying Software Metrics*, IEEE Computer Society Press, 1997) han editado una excelente antología de documentos importantes sobre las métricas del software. Park y otros [PAR96] han desarrollado una guía detallada que proporciona paso a paso sugerencias para instituir un programa de las métricas del software para la mejora del proceso del software.

La hoja informativa *IT Metrics* (Editada por Howard Rubin y publicada por los Servicios de Información Cutter) presenta comentarios Útiles sobre el estado de las métricas del software en la industria. Las revistas *Cutter IT Journal* y *Software Development* tienen habitualmente artículos y características completas dedicadas a las métricas del software.

En Intemet están disponibles una gran variedad de fuentes de información relacionadas con temas del proceso del software y de las métricas del software. Se puede encontrar una lista actualizada con referencias a sitios (páginas) web que son relevantes para el proceso del Software y para las métricas del proyecto en **http://www.pressman5.com.**

CAPÍTULO

5 PLANIFICACIÓN DE PROYECTOS DE SOFTWARE

LA gestión de un proyecto de software comienza con un conjunto de actividades que globalmente se denominan *planificación del proyecto.* Antes de que el proyecto comience, el gestor y el equipo de software deben realizar una estimación del trabajo a realizar, de los recursos necesarios y del tiempo que transcurrirá desde el comienzo hasta el final de su realización. Siempre que estimamos, estamos mirando hacia el futuro y aceptamos resignados cierto grado de incertidumbre.

Aunque la estimación es más un arte que una ciencia, es una actividad importante que no debe llevarse a cabo de forma descuidada. Existen técnicas Útiles para la estimación del esfuerzo y del tiempo. Las métricas del proyecto y del proceso proporcionan una perspectiva histórica y una potente introducción para generar estimaciones cuantitativas. La experiencia anterior (de todas las personas involucradas) puede ayudar en gran medida al desarrollo y revisión de las estimaciones. Y, dado que la estimación es la base de todas las demás actividades de planificación del proyecto, y que sirve como guía para una buena ingeniería del software, no es en absoluto aconsejable embarcarse sin ella.

VISTAZO RÁPIDO

¿Qué es? La planificación de un proyecto de software realmente comprende todas las actividades tratadas en los Capítulos 5 al 9. Sin embargo, en el contexto de este capítulo, la planificación implica la estimación —su intento por determinar cuánto dinero, esfuerzo, recursos, y tiempo supondrá construir un sistema o producto específico de software—.

¿Quién lo hace? Los gestores del software —utilizando la información solicitada a los clientes y a los ingenieros de software y los datos de métricas de software obtenidos de proyectos anteriores—.

¿Por qué es importante? ¿Podría construir una casa sin saber cuanto estaría dispuesto a gastar?. Por supuesto que no, y puesto que la mayoría de los sistemas y productos basados en computadora cuestan considerablemente más que construir una casa grande, podría ser razonable desarrollar y estima antes de empezar a construir el software.

¿Cuáles son los pasos? La estimación comienza con una descripción del ámbito del producto. Hasta que no se «delimita» el ámbito no es posible realizar una estimación con sentido. El problema es entonces descompuesto en un conjunto de problemas de menor tamaño y cada uno de éstos se estima guiándose con datos históricos y con la experiencia. Es aconsejable realizar las estimaciones utilizando al menos dos métodos diferentes (como comprobación). La complejidad y el riesgo del problema se considera antes de realizar una estimación final.

¿Cuál es el producto obtenido? Se obtiene una tabla que indica las tareas a desarrollar, las funciones a implementar, y el coste, esfuerzo y tiempo necesario para la realización de cada una. También se obtiene una lista de recursos necesarios para el proyecto.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Esto es difícil, puesto que realmente no lo sabrá hasta que el proyecto haya finalizado. Sin embargo, si tiene experiencia y sigue un enfoque sistemático, realiza estimaciones utilizando datos históricos sólidos, crea puntos de estimación mediante al menos dos métodos diferentes y descompone la complejidad y riesgo, puede estar seguro de haber acertado plenamente.

5.1 OBSERVACIONES SOBRE LA ESTIMACIÓN

A un destacado ejecutivo se le preguntó una vez por la característica más importante que debe tener un gestor de proyectos. Respondió: « ...una persona con la habilidad de saber qué es lo que va a ir mal antes de que ocurra...» . Debemos añadir: « ...y con el coraje para hacer estimaciones cuando el futuro no está claro...».

La estimación de recursos, costes y planificación temporal de un esfuerzo en el desarrollo de software requiere experiencia, acceder a una buena información histórica y el coraje de confiar en predicciones (medidas) cuantitativas cuando todo lo que existe son datos cualitativos. La estimación conlleva un riesgo inherente[1] y es este riesgo el que lleva a la incertidumbre.



Buenos enfoques de estimación y datos históricos sólidos ofrecen la mejor esperanza de que la realidad puede vencer sobre las demandas imposibles.
Capers Jones

La complejidad del proyecto tiene un gran efecto en la incertidumbre, que es inherente en la planificación. Sin embargo, la complejidad es una medida relativa que se ve afectada por la familiaridad con esfuerzos anteriores. Se podría considerar una aplicación sofisticada de comercio electrónico como «excesivamente compleja» para un desarrollador que haya realizado su primera aplicación. Sin embargo para un equipo de software que desarrolle su enésimo sitio web de comercio electrónico podría considerarse «sumamente fácil» (una de tantas). Se han propuesto una serie de medidas cuantitativas de la complejidad del software (por ejemplo, [ZU597]). Tales medidas se aplican en el nivel de diseño y de codificación, y por consiguiente son difíciles de utilizar durante la planificación del software (antes de que exista un diseño o un código). Sin embargo, al comienzo del proceso de planificación se pueden establecer otras valoraciones de complejidad más subjetivas (por ejemplo, los factores de ajuste de la complejidad del punto de función descritos en el Capítulo 4).

El tamaño del proyecto es otro factor importante que puede afectar a la precisión y a la eficiencia de las estimaciones. A medida que el tamaño aumenta, crece rápidamente[2] la interdependencia entre varios elementos del software. El problema de la descomposición, un enfoque importante hacia la estimación, se hace más difícil porque los elementos descompuestos pueden ser todavía excesivamente grandes. Parafraseando la ley de Murphy: «lo que puede ir mal irá mal», y si hay más cosas que puedan fallar, más cosas fallarán.



La complejidad del proyecto, el tamaño del proyecto y el grado de incertidumbre estructural afectan a la fiabilidad de la estimación.

El grado de incertidumbre estructural tiene también efecto en el riesgo de la estimación. En este contexto, la estructura se refiere al grado en el que los requisitos se han definido, la facilidad con la que pueden subdividirse funciones, y la naturaleza jerárquica de la información que debe procesarse.

La disponibilidad de información histórica tiene una fuerte influencia en el riesgo de la estimación. Al mirar atrás, podemos emular lo que se ha trabajado y mejorar las áreas en donde surgieron problemas. Cuando se dispone de las métricas completas de software de proyectos anteriores (Capítulo 4), se pueden hacer estimaciones con mayor seguridad; establecer planificaciones para evitar dificultades anteriores, y así reducir el riesgo global.



Lo que caracteriza a una inteligencia formada es que puede descansar satisfecha con el grado de precisión que la naturaleza de un asunto permite, y no buscar la exactitud cuando sólo una aproximación de la verdad es posible...
Aristóteles

El riesgo se mide por el grado de incertidumbre en las estimaciones cuantitativas establecidas por recursos, coste y planificación temporal. Si no se entiende bien el ámbito del proyecto o los requisitos del proyecto están sujetos a cambios, la incertidumbre y el riesgo son peligrosamente altos. El planificador del software debería solicitar definiciones completas de rendimiento y de interfaz (dentro de una especificación del sistema). El planificador y, lo que es más importante, el cliente, deben tener presente que cualquier cambio en los requisitos del software significa inestabilidad en el coste y en la planificación temporal.

Sin embargo, un gestor de proyecto no debería obsesionarse con la estimación. Los enfoques modernos de ingeniería del software (por ejemplo, modelos de procesos evolutivos) toman un punto de vista iterativo del desarrollo. En tales enfoques, es posible[3] revisar la estimación (a medida que se conoce más información), y variarla cuando el cliente haga cambios de requisitos.

---

[1] En el Capítulo 6 se presentan técnicas sistemáticas para el análisis del riesgo.

[2] El tamaño se incrementa con frecuencia debido al cambio del ámbito» que ocurre cuando el cliente modifica los requisitos. El incremento del tamaño del proyecto puede tener un impacto geométrico en el coste y en la planificación del proyecto [MAH96].

[3] Esto no significa que siempre sea aceptable políticamente modificar las estimaciones iniciales. Una organización de software madura y sus gestores reconocen que el cambio no es libre. Y sin embargo muchos clientes solicitan (incorrectamente) que una vez realizada la estimación debería mantenerse independientemente de que las circunstancias cambien.

5.2 OBJETIVOS DE LA PLANIFICACIÓN DEL PROYECTO

El objetivo de la planificación del proyecto de software es proporcionar un marco de trabajo que permita al gestor hacer estimaciones razonables de recursos, coste y planificación temporal. Estas estimaciones se hacen dentro de un marco de tiempo limitado al comienzo de un proyecto de software, y deberían actualizarse regularmente a medida que progresa el proyecto. Además, las estimaciones deberían definir los escenarios del «mejor caso» y «peor caso» de forma que los resultados del proyecto puedan limitarse.

El objetivo de la planificación se logra mediante un proceso de descubrimiento de la información que lleve a estimaciones razonables. En las secciones siguientes, se estudian cada una de las actividades asociadas a la planificación del proyecto de software.



Cuanto más sepa, mejor realizará la estimación. Por consiguiente, actualice sus estimaciones a medida que progresa el proyecto.

5.3 ÁMBITO DEL SOFTWARE

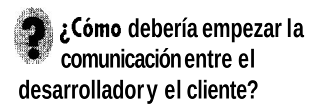
La primera actividad de la planificación del proyecto de software es determinar el ámbito del software. Se deben evaluar la función y el rendimiento que se asignaron al software durante la ingeniería del sistema de computadora (Capítulo 10), para establecer un ámbito de proyecto que no sea ambiguo, ni incomprensible para directivos y técnicos. Se debe *delimitar* la declaración del ámbito del software.

El *ámbito del software* describe el control y los datos a procesar, la función, el rendimiento, las restricciones, las interfaces y la fiabilidad. Se evalúan las funciones descritas en la declaración del ámbito, y en algunos casos se refinan para dar más detalles antes del comienzo de la estimación. Dado que las estimaciones del coste y de la planificación temporal están orientadas a la función, muchas veces es Útil llegar a un cierto grado de descomposición. Las consideraciones de rendimiento abarcan los requisitos de tiempo de respuesta y de procesamiento. Las restricciones identifican los límites del software originados por el hardware externo, por la memoria disponible y por otros sistemas existentes.

5.3.1. Obtención de la información necesaria para el ámbito

Al principio de un proyecto de software las cosas siempre están un poco borrosas. Se ha definido una necesidad y se han enunciado las metas y objetivos básicos, pero todavía no se ha establecido la información necesaria para definir el ámbito (prerrequisito para la estimación).

La técnica utilizada con más frecuencia para acercar al cliente y al desarrollador, y para hacer que comience el proceso de comunicación es establecer una reunión o una entrevista preliminar. La primera reunión entre un ingeniero de software (el analista) y el cliente puede compararse a la primera cita entre adolescentes. Ninguna persona sabe lo que decir o preguntar; ambos están preocupados por si lo que dicen es mal interpretado; ambos están pensando hasta dónde podrían llegar (probablemente los dos tienen aquí diferentes expectativas); ambos quieren quitárselo pronto de encima; pero al mismo tiempo quieren que salga bien.



Sin embargo, se debe iniciar la comunicación. Gause y Weinberg [GAU89] sugieren que el analista comience haciendo *preguntas de contexto libre*. Es decir, una serie de preguntas que lleven a un entendimiento básico del problema, las personas que están interesadas en la solución, la naturaleza de la solución que se desea y la efectividad prevista del primer encuentro.

El primer conjunto de cuestiones de contexto libre se centran en el cliente, en los objetivos globales y en los beneficios. Por ejemplo, el analista podría preguntar:

- ¿Quién está detrás de la solicitud de este trabajo?
- ¿Quién utilizará la solución?
- ¿Cuál será el beneficio económico de una buena solución?
- ¿Hay otro camino para la solución?

Las preguntas siguientes permiten que el analista comprenda mejor el problema y que el cliente exprese sus percepciones sobre una solución:

- ¿Cómo caracterizaría [el cliente] un resultado «correcto» que se generaría con una solución satisfactoria?
- ¿Con qué problema(s) se afrontará esta solución?
- ¿Puede mostrarme (o describirme) el entorno en el que se utilizará la solución?
- ¿Hay aspectos o limitaciones especiales de rendimiento que afecten a la forma en que se aborde la solución?

La última serie de preguntas se centra en la efectividad de la reunión. Gause y Weinberg las llaman «metacuestiones» y proponen la lista siguiente (abreviada):

- ¿Es usted la persona apropiada para responder a estas preguntas? ¿Son «oficiales» sus respuestas?
- ¿Son relevantes mis preguntas para su problema?
- ¿Estoy realizando muchas preguntas?
- ¿Hay alguien más que pueda proporcionar información adicional?
- ¿Hay algo más que debiera preguntarle?

Estas preguntas (y otras) ayudarán a «romper el hielo» y a iniciar la comunicación esencial para establecer el ámbito del proyecto. Sin embargo, una reunión basada en preguntas y respuestas no es un enfoque que haya tenido un éxito abrumador. En realidad, la sesión P&R sólo se debería utilizar para el primer encuentro, reemplazándose posteriormente por un tipo de reunión que combine elementos de resolución de problemas, negociación y especificación.

Los clientes y los ingenieros de software con frecuencia tienen establecido inconscientemente el pensamiento de «nosotros y ellos». En lugar de trabajar como un equipo para identificar y refinar los requisitos, cada uno define su propio «territorio» y se comunica por medio de memorandos, documentos formales de situación, sesiones de preguntas y respuestas e informes. La historia ha demostrado que este enfoque es muy pobre. Abundan los malentendidos, se omite información importante y nunca se establece una relación de trabajo con éxito.

Referencia cruzada

Las técnicas de obtención de requisitos se tratan en el Capítulo 11.

Con estos problemas en mente un grupo de investigadores independientes ha desarrollado un enfoque orientado al equipo para la recopilación de requisitos que pueden aplicarse para ayudar a establecer el ámbito de un proyecto. Las técnicas denominadas *técnicas* para facilitar las especificaciones de la aplicación (TFEA) (*facilitated application specification techniques* [FAST]), pertenecen a un enfoque que alienta a la creación de un equipo compuesto de clientes y de desarolladores que trabajen juntos para identificar el problema, proponer elementos de solución, negociar diferentes enfoques y especificar un conjunto preliminar de requisitos.

5.3.2. Viabilidad

Una vez se ha identificado el ámbito (con la ayuda del cliente), es razonable preguntarse: «¿Podemos construir el software de acuerdo a este ámbito? ¿Es factible el proyecto?». Con frecuencia, las prisas de los ingenieros de software sobrepasan estas preguntas (o están obligados a pasarlas por los clientes o gestores impacientes), solo se tienen en cuenta en un proyecto condenado desde el comienzo. Putnam y Myers [PUT97a] tratan este aspecto cuando escriben:



Hay 150 km a Chicago, tenemos lleno el tanque **de gasolina, medio paquete** de cigarrillos, está **oscuro y llevamos gafas de** sol. ¡Vamos!
The Blues Brothers

...no todo lo imaginable es factible, ni siquiera en el software, fugaz como puede aparecer un forastero. Por el contrario la factibilidad del software tiene cuatro dimensiones sólidas: *Tecnología*—¿Es factible un proyecto técnicamente? ¿Está dentro del estado actual de la técnica? *Financiación*—¿Es factible financieramente? ¿Puede realizarse a un coste asumible **por** la empresa de software y **por** el cliente? *Tiempo*—¿Pueden los proyectos adelantarse a los de la competencia? *Recursos*—¿La organización cuenta con **los** recursos suficientes para tener éxito?

La respuesta es sencilla para algunos proyectos en ciertas áreas, ya que se ha hecho antes algún proyecto de este tipo. **A** las pocas **horas,** *o*, a veces, en pocas semanas de investigación, se puede estar seguro que se puede hacer de nuevo.

Los proyectos de **los** que no se tiene experienciano **son** fáciles. Un equipo puede pasarse varios meses descubriendo cuáles **son** los requisitos principales, y cuáles **son** aquéllos difíciles de implementar para una nueva aplicación. ¿Podría ocurrir que alguno de estos requerimientos presentara algunos riesgos que hicieran inviable el proyecto? ¿Podrían superarse estos riesgos? El equipo de factibilidad debe asumir la arquitectura y el diseño de **los** requerimientos de alto riesgo hasta el punto de poder responder todas estas preguntas. En algunos casos, cuando el equipo proporciona respuestas negativas, esto puede negociarse con una reducción de los requisitos.

Mientras tanto, **los** altos ejecutivos están repicando sus dedos en la mesa. **A** menudo, mueven sus cigarros de forma elegante, pero de forma impaciente y rodeados de humo exclaman ¡Ya es suficiente! ¡Hazlo!

Muchos de estos proyectos que se han aprobado de esta forma aparecen a **los** pocos años en la prensa como proyectos defectuosos.



El estudio de viabilidad es importante, pero las necesidades de la gestión son inclusa más importantes. No es bueno construir un producto o sistema de alta tecnología que en realidad nadie quiera.

Putnam y Myers sugieren, de forma acertada, que el estudio del ámbito no es suficiente. Una vez que se ha comprendido el ámbito, tanto el equipo de desarrollo como el resto deben trabajar para determinar si puede ser construido dentro de las dimensiones reflejadas anteriormente. Esto es crucial, aunque es una parte del proceso de estimación pasada por alto a menudo.

5.3.3. Un ejemplo de ámbito

La comunicación con el cliente lleva a una definición del control y de los datos procesados, de las funciones que deben ser implementadas, del rendimiento y restricciones que delimitan el sistema, y de la información relacionada. Como ejemplo, consideremos el software

que se debe desarrollar para controlar un sistema de clasificación de cinta transportadora (SCCT). La especificación del ámbito del SCCT es la siguiente:

*El sistema **de** clasificación de cinta transportadora (SCCT)* clasifica las cajas que se mueven por una cinta transportadora. Cada caja estará identificada por un código de barras que contiene un número de pieza y se clasifica en uno de seis compartimentos **al** final de la cinta. Las cajas pasarán por una estación de clasificación que consta de un lector de código de barras y un **PC.** **El** PC de la estación de clasificación está conectado a un mecanismo de maniobra que clasifica las cajas en **los** compartimentos. Las cajas pasan en orden aleatorio y están espaciadas uniformemente. La cinta se mueve a cinco pies por minuto. En la Figura 5.1 está representado esquemáticamente el **SCCT.**

El software del **SCCT** debe recibir información de entrada de un lector de código de barras a intervalos de tiempo que se ajusten **a** la velocidad de la cinta transportadora. **Los** datos del código de barras se decodifican al formato de identificación de caja. El software llevará a cabo una inspección en la base de datos de números de piezas que contiene un máximo de 1000 entradas **para** determinar la posición del compartimento adecuada para la caja que se encuentre actualmente en el lector (estación de clasificación). La posición correcta del compartimento se pasará **a** un mecanismo de maniobra de ordenación que sitúa las cajas en el **lugar** adecuado. Se mantendrá una lista con los compartimentos destino de cada caja para su **posterior** recuperación e informe. **El** software del **SCCT** recibirá también entrada de un tacómetro de **pulsos** que se utilizará para sincronizar la señal de control del mecanismo de maniobra. Basándonos en el número de **pulsos** que **se** generen entre la estación de clasificación y el mecanismo de maniobra, el software producirá una señal de control para que la maniobra sitúe adecuadamente la caja.

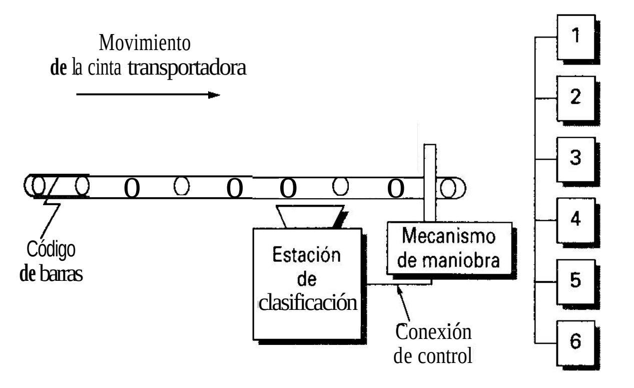


FIGURA 5.1. Un sistema de clasificación de cinta transportadora.

El planificador del proyecto examina la especificación del ámbito **y** extrae todas las funciones importantes del software. Este proceso, denominado *descomposición*, se trató en el Capítulo *3* y produce como resultado las funciones siguientes[4]:

- Lectura de la entrada del código de barras.
- Lectura del tacómetro de pulsos.
- Descodificación de los datos del código de pieza.
- Búsqueda en la base datos.
- Determinar la posición del compartimento.
- Producción de la señal de control para el mecanismo de maniobra.
- Mantener una lista de los destinos de las cajas

En este caso, el rendimiento está determinado por la velocidad de la cinta transportadora. Se tiene que terminar el procesamiento de cada caja antes de que llegue la siguiente caja al lector de código de barras. El software del SCCT está limitado por el hardware al que tiene que acceder —el lector de código de barras, el mecanismo de maniobra, la computadora personal (PC)—, la memoria disponible y la configuración global de la cinta transportadora (cajas uniformemente espaciadas).



*Ajuste lo **estimación** poro **reflejar** los requisitos del rendimiento y los **restricciones** del **diseño** difíciles, incluso **si** el **ámbito** es sencillo.*

La función, el rendimiento y las restricciones se evalúan a la vez. Una misma función puede producir una diferencia de un orden de magnitud en el esfuerzo de desarrollo cuando se considera en un contexto con diferentes límites de rendimiento. El esfuerzo y los costes requeridos para desarrollar el software SCCT serían drásticamente diferentes si la función fuera la misma (por ejemplo: la cinta transportadora siguiese colocando cajas en contenedores), pero el rendimiento variará. Por ejemplo, si la velocidad media de la cinta transportadora aumentara en un factor de 10 (rendimiento) y las cajas no estuvieran uniformemente espaciadas (una restricción), el software podría ser considerablemente más complejo —requiriendo, por tanto, un mayor esfuerzo de desarrollo—. La función, el rendimiento y las restricciones están íntimamente relacionadas.

CLAVE

La consideración del ámbito del software debe contener una evaluación de todas las interfaces externas.

El software interactúa con otros elementos del sistema informático. El planificador considera la naturaleza y la complejidad de cada interfaz para determinar cualquier efecto sobre los recursos, los costes y la planificación temporal del desarrollo. El concepto de interfaz abarca lo siguiente: (1) hardware (por ejemplo: procesador, periféricos) que ejecuta el software y los dispositivos (por ejemplo: máquinas, pantallas) que están controlados indirectamente por el software; (2) softwa-

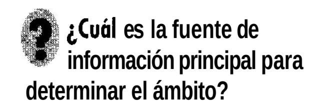
[4] En realidad, la descomposición funcional se hace durante la ingeniería del sistema (Capítulo 10). El planificador utiliza la información obtenida a partir de la especificación del sistema para definir funciones del software.

re ya existente (por ejemplo, rutinas de acceso a una base de datos, componentes de software reutilizables, sistemas operativos) que debe ser integrado con el nuevo software; (3) personas que hacen uso del software por medio del teclado (terminales) u otros dispositivos de entrada/salida; (4) procedimientos que preceden o suceden al software en una secuencia de operaciones. En cada caso, debe comprenderse claramente la información que se transfiere a través de la interfaz.

Si se ha desarrollado adecuadamente la *especificación del sistema* (véase el Capítulo 10), casi toda la información requerida para la descripción del ámbito del software estará disponible y documentada antes de que comience la planificación del proyecto de software. En los casos en los que no haya sido desarrollada la especificación, el planificador debe hacer el papel del analista de sistemas para determinar las características y las restricciones que influirán en las tareas de estimación.

5.4 RECURSOS

La segunda tarea de la planificación del desarrollo de software es la estimación de los recursos requeridos para acometer el esfuerzo de desarrollo de software. La Figura 5.2 ilustra los recursos de desarrollo en forma de pirámide. En la base de la pirámide de recursos se encuentra el *entorno de desarrollo* —herramientas de hardware y software— que proporciona la infraestructura de soporte al esfuerzo de desarrollo. En un nivel más alto se encuentran los *componentes de software reutilizables* —los bloques de software que pueden reducir drásticamente los costes de desarrollo y acelerar la entrega—. En la parte más alta de la pirámide está el recurso primario —*el personal*—. Cada recurso queda especificado mediante cuatro características: descripción del recurso, informe de disponibilidad, fecha cronológica en la que se requiere el recurso, tiempo durante el que será aplicado el recurso. Las dos últimas características pueden verse como una ventana temporal. La disponibilidad del recurso para una ventana específica tiene que establecerse lo más pronto posible.



5.4.1. Recursos humanos

El encargado de la planificación comienza elevando el ámbito y seleccionando las habilidades que se requieren para llevar a cabo el desarrollo. Hay que especificar tanto la posición dentro de la organización (por ejemplo: gestor, ingeniero de software experimentado, etc.) como la especialidad (por ejemplo: telecomunicaciones, bases de datos, cliente/servidor). Para proyectos relativamente pequeños (una persona-año o menos) una sola persona puede llevar a cabo todos los pasos de ingeniería del software, consultando con especialistas siempre que sea necesario.

El número de personas requerido para un proyecto de software sólo puede ser determinado después de hacer una estimación del esfuerzo de desarrollo (por ejemplo, personas-mes). Estas técnicas de estimación del esfuerzo se estudiarán después en este mismo capítulo.

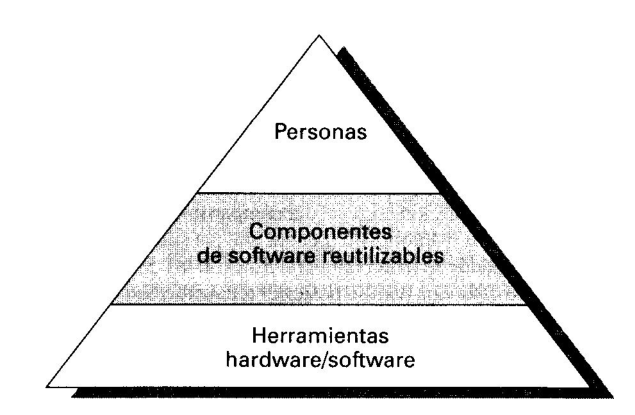


FIGURA 5.2. Recursos del proyecto.

5.4.2. Recursos de software reutilizables

La ingeniería del software basada en componentes (ISBC)[5] destaca la reutilización -esto es, la creación y la reutilización de bloques de construcción de software [HOO91]—. Dichos bloques de construcción, llamados componentes, deben establecerse en catálogos para una consulta más fácil, estandarizarse para una fácil aplicación y validarse para una fácil integración.

Referencia cruzada
El papel que juegan las personas implicadas en el software y las organizaciones del equipo al que pertenecen se tratan en el Capítulo 3.

Bennatan [BEN92] sugiere cuatro categorías de recursos de software que se deberían tener en cuenta a medida que se avanza con la planificación:

Componentes ya desarrollados. El software existente se puede adquirir de una tercera parte o provenir de uno desarrollado internamente para un proyecto anterior. Llamados componentes CCYD (componentes comercialmente ya desarrollados), estos componentes están listos para utilizarse en el proyecto actual y se han validado totalmente.

Componentes ya experimentados. Especificaciones, diseños, código o datos de prueba existentes

[5] La ingeniería del software basada en componentes se trata con detalle en el Capítulo 27.

desarrollados para proyectos anteriores que son similares al software que se va a construir para el proyecto actual. Los miembros del equipo de software actual ya han tenido la experiencia completa en el área de la aplicación representada para estos componentes. Las modificaciones, por tanto, requeridas para componentes de total experiencia, tendrán un riesgo relativamente bajo.



CLAVE

Para que la reutilización sea eficiente, los componentes de software deben estar *catalogados*, estandarizados y validados.

Componentes con experiencia parcial. Especificaciones, diseños, código o datos de prueba existentes desarrollados para proyectos anteriores que se relacionan con el software que se va a construir para el proyecto actual, pero que requerirán una modificación sustancial. Los miembros del equipo de software actual han limitado su experiencia sólo al área de aplicación representada por estos componentes. Las modificaciones, por tanto, requeridas para componentes de experiencia parcial tendrán bastante grado de riesgo.

Componentes nuevos. Los componentes de software que el equipo de software debe construir específicamente para las necesidades del proyecto actual.

¿Qué aspectos debemos considerar cuando planificamos la reutilización de componentes de software existentes?

Deberían ser consideradas las directrices siguientes por el planificador de software cuando los componentes reutilizables se especifiquen como recurso:

1. Si los componentes ya desarrollados cumplen los requisitos del proyecto, adquiéralos. El coste de la adquisición y de la integración de los componentes ya desarrollados serán casi siempre menores que el coste para desarrollar el software equivalente[6]. Además, el riesgo es relativamente bajo.
2. Si se dispone de componentes ya experimentados, los riesgos asociados a la modificación y a la integración generalmente se aceptan. El plan del proyecto debería reflejar la utilización de estos componentes.
3. Si se dispone de componentes de experiencia parcial para el proyecto actual, su uso se debe analizar con detalle. **Si** antes de que se integren adecuadamente los componentes con otros elementos del software se requiere una gran modificación, proceda cuidadosamente - e l riesgo es alto—. El coste de modificar los componentes de experiencia parcial algunas veces puede ser mayor que el coste de desarrollar componentes nuevos.

De forma irónica, a menudo se descuida la utilización de componentes de software reutilizables durante la planificación, llegando a convertirse en la preocupación primordial durante la fase de desarrollo del proceso de software. Es mucho mejor especificar al principio las necesidades de recursos del software. De esta forma se puede dirigir la evaluación técnica de alternativas y puede tener lugar la adquisición oportuna.

5.4.3. Recursos de entorno

El entorno es donde se apoya el proyecto de software, llamado a menudo *entorno de ingeniería del software (EIS)*, incorpora hardware y software. El hardware proporciona una plataforma con las herramientas (software) requeridas para producir los productos que son el resultado de una buena práctica de la ingeniería del software[7]. Como la mayoría de las organizaciones de software tienen muchos aspectos que requieren acceso a **EIS,** un planificador de proyecto debe determinar la ventana temporal requerida para el hardware y el software, y verificar que estos recursos estarán disponibles.

Cuando se va a desarrollar un sistema basado en computadora (que incorpora hardware y software especializado), el equipo de software puede requerir acceso a los elementos en desarrollo por otros equipos de ingeniería. Por ejemplo, el software para un *control numérico (CN)* utilizado en una clase de máquina herramienta puede requerir una máquina herramienta específica (por ejemplo, el CN de un torno) como parte del paso de prueba de validación; un proyecto de software para el diseño de páginas avanzado puede necesitar un sistema de composición fotográfica o escritura digital en alguna fase durante el desarrollo. Cada elemento de hardware debe ser especificado por el planificador del proyecto de software.

[6] Cuando los componentes de software existentes se utilizan durante un proyecto, la reducción del coste global puede ser importante. En efecto, los datos de la industria indican que el coste, el tiempo de adquisición y el número de defectos entregados al cliente se reducen.

[7] Otro hardware —el entorno destino— es la coinnputadora sobre la que se va a ejecutar el software al entregarse al usuario final.

5.5 ESTIMACIÓN DEL PROYECTO DE SOFTWARE

Al principio, el coste del software constituía un pequeño porcentaje del coste total de los sistemas basados en computadora. Un error considerable en las estimaciones del coste del software tenía relativamente poco impacto. Hoy en día, el software es el elemento más caro de la mayoría de los sistemas informáticos. Para sistemas complejos, personalizados, un gran error en la estimación del coste puede ser lo que marque la diferencia entre beneficios y pérdidas. Sobrepasarse en el coste puede ser desastroso para el desarrollador.

La estimación del coste y del esfuerzo del software nunca será una ciencia exacta. Son demasiadas las variables —humanas, técnicas, de entorno, políticas— que pueden afectar al coste final del software y al esfuerzo aplicado para desarrollarlo. Sin embargo, la estimación del proyecto de software puede dejar de ser un oscuro arte para convertirse en una serie de pasos sistemáticos que proporcionen estimaciones con un grado de riesgo aceptable.



En una era de *outsourcing* y competencia creciente, la capacidad para estimar de un modo más exacto....se ha convertido en un factor crítico de supervivencia para muchos grupos TI.
Rob Thomsett

Para realizar estimaciones seguras de costes y esfuerzos tenemos varias opciones posibles:

1. Dejar la estimación para más adelante (obviamente, ¡podemos realizar una estimación al cien por cien fiable tras haber terminado el proyecto!).
2. Basar las estimaciones en proyectos similares ya terminados.
3. Utilizar «técnicas de descomposición» relativamente sencillas para generar las estimaciones de coste y de esfuerzo del proyecto.
4. Utilizar uno o más modelos empíricos para la estimación del coste y esfuerzo del software.

Desgraciadamente, la primera opción, aunque atractiva, no es práctica. Las estimaciones de costes han de ser proporcionadas *a priori*. Sin embargo, hay que reconocer que cuanto más tiempo esperemos, más cosas sabremos, y cuanto más sepamos, menor será la probabilidad de cometer serios errores en nuestras estimaciones.

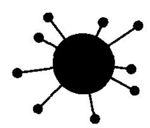
La segunda opción puede funcionar razonablemente bien, si el proyecto actual es bastante similar a los esfuerzos pasados y si otras influencias del proyecto (por ejemplo: el cliente, las condiciones de gestión, el **EIS** [Entorno de Ingeniería del software], las fechas límites) son similares. Por desgracia, la experiencia anterior no ha sido siempre un buen indicador de futuros resultados.

Las opciones restantes son métodos viables para la estimación del proyecto de software. Desde un punto de vista ideal, se deben aplicar conjuntamente las técnicas indicadas; usando cada una de ellas como comprobación de las otras. Las *técnicas de descomposición* utilizan un enfoque de «divide y vencerás)) para la estimación del proyecto de software. Mediante la descomposición del proyecto en sus funciones principales y en las tareas de ingeniería del software correspondientes, la estimación del coste y del esfuerzo puede realizarse de una forma escalonada idónea. Se pueden utilizar los *modelos empíricos de estimación* como complemento de las técnicas de descomposición, ofreciendo un enfoque de estimación potencialmente valioso por derecho propio. Cada modelo se basa en la experiencia (datos históricos) y toma como base:

$$d = f(v_i)$$

donde ***d*** es uno de los valores estimados (por ejemplo, esfuerzo, coste, duración del proyecto) y los v_i, son determinados parámetros independientes (por ejemplo, LDC o PF estimados).

Las *herramientas automáticas de estimación* implementan una o varias técnicas de descomposición o modelos empíricos. Cuando se combinan con una interfaz gráfica de usuario, las herramientas automáticas son una opción atractiva para la estimación. En sistemas de este tipo, se describen las características de la organización de desarrollo (por ejemplo, la experiencia, el entorno) y el software a desarrollar. De estos datos se obtienen las estimaciones de coste y de esfuerzo.



Herramientas CASE.

Cada una de las opciones viables para la estimación de costes del software, sólo será buena si los datos históricos que se utilizan como base de la estimación son buenos. Si no existen datos históricos, la estimación del coste descansará sobre una base muy inestable. En el Capítulo 4 examinamos las características de los datos de productividad del software y cómo pueden utilizarse como base histórica para la estimación.

5.6 TÉCNICAS DE DESCOMPOSICIÓN

La estimación de proyectos de software es una forma de resolución de problemas y, en la mayoría de los casos, el problema a resolver (es decir, desarrollar estimaciones de coste y de esfuerzo para un proyecto de software) es demasiado complejo para considerarlo como un todo. Por esta razón, descomponemos el problema, volviéndolo a definir como un conjunto de pequeños problemas (esperando que sean más manejables).

En el Capítulo *3*, se estudió el enfoque de descomposición desde dos puntos de vista diferentes: descomposición del problema y descomposición del proceso. La estimación hace uso de una o ambas formas de particionamiento. Pero antes de hacer una estimación, el planificador del proyecto debe comprender el ámbito del software a construir y generar una estimación de su «tamaño».

5.6.1. Tamaño del software

La precisión de una estimación del proyecto de software se predice basándose en una serie de cosas: (1) el grado en el que el planificador ha estimado adecuadamente el tamaño del producto a construir; (2) la habilidad para traducir la estimación del tamaño en esfuerzo humano, tiempo y dinero (una función de la disponibilidad de métricas fiables de software de proyectos anteriores); (3) el grado en el que el plan del proyecto refleja las habilidades del equipo de software, y (4) la estabilidad de los requisitos del software y el entorno que soporta el esfuerzo de la ingeniería del software.



CLAVE

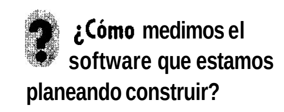
El «tamaño» del software a construir puede estimarse utilizando una medida directa, LDC, o una medida indirecta, PF.

En esta sección, consideramos el problema del *tamaño del software*. Puesto que una estimación del proyecto es tan buena como la estimación del tamaño del trabajo que va a llevarse a cabo, el tamaño representa el primer reto importante del planificador de proyectos. En el contexto de la planificación de proyectos, el tamaño se refiere a una producción cuantificable del proyecto de software. Si se toma un enfoque directo, el tamaño se puede medir en LDC. Si se selecciona un enfoque indirecto, el tamaño se representa como PF.

Putnam y Myers [PUT92] sugieren cuatro enfoques diferentes del problema del tamaño:

Tamaño en «lógica difusa». Este enfoque utiliza las técnicas aproximadas de razonamiento que son la piedra angular de la lógica difusa. Para aplicar este enfoque, el planificador debe identificar el tipo de aplicación, establecer su magnitud en una escala cuantitativa y refinar la magnitud dentro del rango original. Aunque se puede utilizar la experiencia personal, el planificador también debería tener acceso a una base de datos histórica de proyectos[8] para que las estimaciones se puedan comparar con la experiencia real.

Tamaño en punto de función. El planificador desarrolla estimaciones de características del dominio de información estudiadas en el Capítulo 4.



Tamaño de componentes estándar. El software se compone de un número de «componentes estándar» que son genéricos para un área en particular de la aplicación. Por ejemplo, los componentes estándar para un sistema de información son: subsistemas, módulos, pantallas, informes, programas interactivos, programas por lotes, archivos, LDC e instrucciones para objetos. El planificador de proyectos estima el número de incidencias de cada uno de los componentes estándar, y utiliza datos de proyectos históricos para determinar el tamaño de entrega por componente estándar. Para ilustrarlo, considere una aplicación de sistemas de información. El planificador estima que se generarán 18 informes. Los datos históricos indican que por informe se requieren 967 líneas de Cobol [PUT92]. Esto permite que el planificador estime que se requieren 17.000 LDC para el componente de informes. Para otros componentes estándar se hacen estimaciones y cálculos similares, y se producen resultados combinados de valores de tamaños (ajustados estadísticamente).

Tamaño del cambio. Este enfoque se utiliza cuando un proyecto comprende la utilización de software existente que se debe modificar de alguna manera como parte de un proyecto. El planificador estima el número y tipo (por ejemplo: reutilización, añadir código, cambiar código, suprimir código) de modificaciones que se deben llevar a cabo. Mediante una «proporción de esfuerzo» [PUT92] para cada tipo de cambio, se puede estimar el tamaño del cambio.

[8] Consulte la Sección 5.9 que trata brevemente las herramientas de estimación que utilizan una base de datos histórica y otras técnicas de tamaño estudiadas en esta sección.

Putnam y Myers sugieren que los resultados de cada uno de los métodos de tamaño señalados anteriormente se combinan estadísticamente para crear una *estimación de tres puntos o del valor esperado.* Esto se lleva a cabo desarrollando valores de tamaño optimistas (bajos), y más probables, y pesimistas (altos), y se combinan utilizando la Ecuación (5.1) que se describe en la sección siguiente.

5.6.2. Estimación basada en el problema

En el Capítulo 4, las líneas de código (LDC) y los puntos de función (PF) se describieron como medidas básicas a partir de las que se pueden calcular métricas de productividad. Los datos de LDC y PF se utilizan de dos formas durante la estimación del proyecto de software: (1) como una variable de estimación que se utiliza para «dimensionar» cada elemento del software, y (2) como métricas de línea base recopiladas de proyectos anteriores y utilizadas junto con variables de estimación para desarrollar proyecciones de coste y de esfuerzo.

Las estimaciones de LDC y PF son técnicas de estimación distintas. **A** pesar de que ambas tienen varias características **en** común. El planificador del proyecto comienza con un enfoque limitado para el ámbito del software y desde este estado intenta descomponer el software en funciones que se pueden estimar individualmente. Para cada función entonces se estiman las LDC y el PF (la variable de estimación). De forma alternativa, el planificador puede seleccionar otro componente para dimensionar clases u objetos, cambios o procesos de gestión en los que puede tener impacto.

¿Qué tienen en común la estimación orientada a PF y a LDC?

Las métricas de productividad de línea base (por ejemplo: LDC/pm o PF/pm[9]) se aplican entonces para la variable de estimación adecuada y se extrae el coste o el esfuerzo de la función. Las estimaciones de función se combinan para producir una estimación global del proyecto entero.



Cuando se comparan métricas de productividad para proyectos, esté seguro de establecer una clasificación de los tipos de proyectos. Esto le permitirá calcular promedios para dominios específicos y la estimación será más exacta.

Es importante señalar, sin embargo, que hay una sustancial diversidad en métricas de productividad, haciendo sospechar que se utilice únicamente una métrica de productividad de línea base. En general, el dominio del proyecto debería calcular las medias de LDC/pm o PF/pm. Es decir, los proyectos se deberían agrupar por tamaño de equipo, área de aplicación, complejidad y otros parametros relevantes. Entonces se deberían calcular las medias del dominio local. Cuando se estima un proyecto nuevo, primero se debería asignar a un dominio, y a continuación utilizar la media del dominio adecuado para la productividad al generar la estimación. Las técnicas de estimación de LDC y PF difieren en el nivel de detalle que se requiere para la descomposición y el objetivo de la partición. Cuando se utiliza LDC como variable de estimación, la descomposición[10] es absolutamente esencial y con frecuencia se toman para considerables niveles de detalle. El enfoque de descomposición siguiente ha sido adaptado por Phillips [PHI98][11]:

CLAVE

Para estimaciones de LDC, la descomposición se centra en las funciones del software.

```
definir el ámbito del producto;
identificar funciones descomponiendo el ámbito;
hacer mientras haya funciones
    seleccionar una función_j
    asignar todas las funciones a la lista de subfunciones;
    hacer mientras haya subfunciones
        seleccionar una subfunción_k
        si subfunción_k = subfunción_d descrita en una base
                de datos histórica
        entonces  anotar datos históricos del coste,
                  esfuerzo, tamaño, (LDC o PF) para
                  la subfunción_d;
                  ajustar datos históricos del coste,
                  esfuerzo, tamaño basados en cualquier diferencia:
                  use datos del coste, esfuerzo, tamaño ajustados para obtener una estimación parcial, E_p;
                  estimación proyecto = suma de
                  { E_p };
        sino      si se puede estimar coste, esfuerzo,
                  tamaño (LDC o PF) para subfunción_k
                  entonces obtener estimación parcial,

                  estimación proyecto = suma de
                  { E_p };
                  sino subdividir subfunción_k en subfunciones más pequeñas;
                  añadirlas a la lista de subfunciones;
                    fin si
            fin si
        fin hacer
fin hacer
```

[9] El acrónimo pm hace referencia a personas-mes

[10] En general se descomponen las funciones del problema. Sin embargo. se puede utilizar una lista de componentes estándar (Sección 5.6.1).

[11] El lenguaje informal de diseño del proceso señalado aquí se expone para ilustrar el enfoque general para el dimensionamiento. No tiene en cuenta ninguna eventualidad lógica.

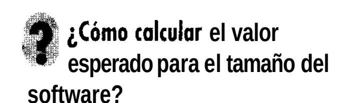
El enfoque de descomposición visto anteriormente asume que todas las funciones pueden descomponerse en subfunciones que podrán asemejarse a entradas de una base de datos histórica. Si este no es el caso, deberá aplicarse otro enfoque de dimensionamiento. Cuanto más grande sea el grado de particionamiento, más probable será que pueda desarrollar estimaciones de LDC más exactas.

Para estimaciones de PF, la descomposición funciona de diferente manera. En lugar de centrarse en la función, se estiman cada una de las características del dominio de información —entradas, salidas, archivos de datos, peticiones, e interfaces externas— y los catorce valores de ajuste de la complejidad estudiados en el Capítulo **4.** Las estimaciones resultantes se pueden utilizar para derivar un valor de PF que se pueda unir a datos pasados **y** utilizar para generar una estimación.

CLAVE

Para estimaciones de PF, la descomposición se centra en las características del dominio de información.

Con independencia de la variable de estimación que se utilice, el planificador del proyecto comienza por estimar un rango de valores para cada función o valor del dominio de información. Mediante los datos históricos o (cuando todo lo demás falla) la intuición, el planificador estima un valor de tamaño optimista, más probable y pesimista para cada función, o cuenta para cada valor del dominio de información. Al especificar un rango de valores, se proporciona una indicación implícita del grado de incertidumbre.



Entonces se calcula un valor de tres puntos o esperado. El *valor esperado* de la variable de estimación (tamaño), *VE*, se puede calcular como una media ponderada de las estimaciones optimistas (S_{opt}), las más probables (S_m), y las pesimistas (S_{pess}). Por ejemplo,

$$VE = (S_{opt} + 4S_m + S_{pess}) / 6 \qquad (5.1)$$

da crédito a la estimación «más probable» y sigue una distribución de probabilidad beta. Se asume que existe una probabilidad muy pequeña en donde el resultado del tamaño real quedará fuera de los valores pesimistas y optimistas.

Una vez que se ha determinado el valor esperado de la variable de estimación, se aplican datos históricos de productividad de LDC y PF. ¿Son correctas las estimaciones? La única respuesta razonable a esto es: «No podemos estar seguros». Cualquier técnica de estimación, no importa lo sofisticada que sea, se debe volver a comprobar con otro enfoque. Incluso entonces, va a prevalecer el sentido común y la experiencia.

5.6.3. Un ejemplo de estimación basada en LDC

Como ejemplo de técnicas de estimación de LDC y PF, tomemos en consideración un paquete de software a desarrollarse para una aplicación de diseño asistido por computadora (computer-aided design, CAD) de componentes mecánicos. Una revisión de la *especificación del sistema* indica que el software va a ejecutarse en una estación de trabajo de ingeniería y que debe interconectarse con varios periféricos de gráficos de computadora entre los que se incluyen un ratón, un digitalizador, una pantalla a color de alta resolución y una impresora láser.

Utilizando como guía una *especificación de sistema*, se puede desarrollar una descripción preliminar del ámbito del software:

> El software de CAD aceptará datos geométricos de dos y tres dimensiones por parte del ingeniero. El ingeniero se interconectará y controlará el sistema de CAD por medio de una interfaz de usuario que va a exhibir las características de un buen diseño de interfaz hombre-máquina. Una base de datos de CAD contiene todos los datos geométricos y la información de soporte. Se desarrollarán módulos de análisis de diseño para producir la salida requerida que se va a visualizar en varios dispositivos gráficos. El software se diseñará para controlar e interconectar dispositivos periféricos entre los que se incluyen un ratón, un digitalizador, una impresora láser y un trazador gráfico (*plotter*).

La descripción anterior del ámbito es preliminar —no está limitada—. Para proporcionar detalles concretos y un límite cuantitativo se tendrían que ampliar todas las descripciones. Por ejemplo, antes de que la estimación pueda comenzar, el planificador debe determinar **qué** significa «características de un buen diseño de interfaz hombre-máquina» y cuál será el tamaño y la sofisticación de la «base de datos de CAD».

Para estos propósitos, se asume que ha habido más refinamiento **y** que se identifican las funciones de software siguientes:

- interfaz de usuario y facilidades de control (IUFC),
- análisis geométrico de dos dimensiones (AG2D),
- análisis geométrico de tres dimensiones (AG3D),
- gestión de base de datos (GBD),
- facilidades de presentación gráfica de computadora (FPGC),
- control de periféricos (CP),
- módulos de análisis del diseño (MAD).



Muchas aplicaciones modernos residen en uno red o son porte de uno arquitectura cliente/servidor. Por consiguiente, esté seguro que sus estimaciones contienen el esfuerzo requerido para el desarrollo del software de la «infraestructura».

Siguiendo la técnica de descomposición de LDC, se desarrolla la tabla de estimación mostrada en la Figura 5.3. Se desarrolla un rango de estimaciones de LDC para

cada función. Por ejemplo, el rango de estimaciones de LDC de la función del análisis geométrico de tres dimensiones es: optimista -4.600 LDC—; más probable —6.900—, y pesimista —8.600 LDC—. Aplicando la Ecuación (5.1), el valor esperado de la función del análisis geométrico es 6.800 LDC. Este número se introduce en la tabla. De forma similar se derivan otras estimaciones. Sumando en vertical la columna estimada de LDC, se establece una estimación de 33.200 líneas de código para el sistema de CAD.



No sucumba a la tentación de utilizar este resultado como estimación. Debería obtener otra estimación utilizando un método diferente.

Una revisión de datos históricos indica que la productividad media de la organización para sistemas de este tipo es de 620 LDC/pm. Según una tarifa laboral de E8.000 por mes, el coste por línea de código es aproximadamente de £13,00. Según la estimación de LDC y los datos de productividad históricos, el coste total estimado del proyecto es de £431.000 y el esfuerzo estimado es de 54 personas-mes[12].

Función	LDC estimada
Interface del usuario y facilidades de control (IUFC)	2.300
Análisis geométrico de dos dimensiones (AG2D)	5.300
Análisis geométrico de tres dimensiones (A32D)	6.800
Gestión de base de datos (GBD)	3.350
Facilidades de presentación gráfica de computadora (FPGC)	4.950
Control de periféricos (CP)	2.100
Módulos de análisis del diseño (MAD)	8.400
Líneas de códigos estimadas	33.200

FIGURA 5.3. Tabla de estimación para el método de LDC.

5.6.4. Un ejemplo de estimación basada en PF

La descomposición de estimación basada en PF se centra en los valores de dominio de información, y no en las funciones del software. Teniendo en cuenta la tabla de cálculos de punto de función presentada en la Figura 5.4, el planificador del proyecto estima las entradas, salidas, peticiones, archivos e interfaces externas del software de CAD. Para los propósitos de esta estimación, el factor de ponderación de la complejidad se asume como la media. La Figura 5.4 presenta los resultados de esta estimación.



Se puede obtener información sobre las herramientas de estimación del coste mediante PF en **www.spr.com**

Valor de dominio de información	Opt.	Probable	Pes.	Cuenta est.	Peso	Cuenta PF
Número de entradas	20	24	30	24	4	97
Número de salidas	12	15	22	16	5	78
Número de peticiones	16	22	28	22	5	88
Número de archivos	4	4	5	4	10	42
Número de interfaces externas	2	2	3	2	7	15
Cuenta total						320

FIGURA 5.4. Estimación de los valores del dominio de información.

Como se describe en el Capítulo 4, se estima cada uno de los factores de ponderación de la complejidad, y se calcula el factor de ajuste de la complejidad:

Factor	Valor
Copia de seguridad y recuperación	4
Comunicaciones de datos	2
Proceso distribuido	0
Rendimiento crítico	4
Entorno operativo existente	3
Entrada de datos en línea (on-line)	4
Transacciones de entrada en múltiples pantallas	5
Archivos maestros actualizados en línea (on-line)	3
Complejidad de valores del dominio de información	5
Complejidad del procesamiento interno	5
Código diseñado para la reutilización	4
Conversión/instalación en diseño	3
Instalaciones múltiples	5
Aplicación diseñada para el cambio	5
Factor de ajuste de la complejidad	1,17

Finalmente, se obtiene el número estimado de PF:

$$PF_{estimado} = \text{cuenta total} \times [0,65 + 0,01 \times \Sigma(F_i)]$$

$$PF_{estimado} = 375$$

La productividad media organizacional para sistemas de este tipo es de 6,5 PF/pm. Según la tarifa laboral de E8.000 por mes, el coste por PF es aproximadamente de E1.230. Según la estimación de LDC y los datos de productividad históricos, el coste total estimado del proyecto es de £461.000, y el esfuerzo estimado es de 58 personas/mes.

[12] La estimación se redondea acercándose lo más posible a E1.000 y persona–mes.

5.6.5. Estimación basada en el proceso

La técnica más común para estimar un proyecto es basar la estimación en el proceso que se va a utilizar. **Es** decir, el proceso se descompone en un conjunto relativamente pequeño de actividades o tareas, y en el esfuerzo requerido para llevar a cabo la estimación de cada tarea.

Al igual que las técnicas basadas en problemas, la estimación basada en el proceso comienza con un esbozo de las funciones del software obtenidas a partir del ámbito del proyecto. Para cada función se debe llevar a cabo una serie de actividades del proceso de software. Las funciones y las actividades del proceso de software relacionadas se pueden representar como parte de una tabla similar a la presentada en la Figura 3.2.

Referencia cruzada

El marco de trabajo común del proceso (MCP) se estudio en el Capítulo 2.

Una vez que se mezclan las funciones del problema y las actividades del proceso, el planificador estima el esfuerzo (por ejemplo: personas-mes) que se requerirá para llevar a cabo cada una de las actividades del proceso de software en cada función. Estos datos constituyen la matriz central de la tabla de la Figura 3.2. Los índices de trabajo medios (es decir, esfuerzo coste/unidad) se aplican entonces al esfuerzo estimado a cada actividad del proceso. Es muy probable que en cada tarea varíe el índice de trabajo. El personal veterano se involucra de lleno con las primeras actividades **y** generalmente es mucho más caro que el personal junior, quienes están relacionados con las tareas de diseño posteriores, con la generación del código y con las pruebas iniciales.

Actividad →	CC	Planificación	Análisis de riesgo	Ingenieria		Construcción entrega		EC	Totales
Tarea →				Análisis	Diseño	Código	Prueba		
Función ↓									
IUFC				0.50	2.50	0.40	5.00	n/a	8.40
AG2D				0.75	4.00	0.60	2.00	n/a	7.35
AG3D				0.50	4.00	1.00	3.00	n/a	8.50
FPGC				0.50	3.00	1.00	1.50	n/a	6.00
GBD				0.50	3.00	0.75	1.50	n/a	5.75
CP				0.25	2.00	0.50	1.50	n/a	4.25
MAD				0.50	2.00	0.50	2.00	n/a	5.00
Totales	10.251	0.25	0.25	3.50	120.50	4.75	16.50		46.00
% Esfuerzo	1%	1%	1%	8%	45%	10%	36%		

CC = Comunicación cliente EC = Evaluación cliente

FIGURA 5.5. Tabla de estimación basada en el proceso.

Como último paso se calculan los costes y el esfuerzo de cada función, y la actividad del proceso de software. **Si** la estimación basada en el proceso se realiza independientemente de la estimación de LDC o PF, ahora tendremos dos o tres estimaciones del coste y del esfuerzo que se pueden comparar y evaluar. Si ambos tipos de estimaciones muestran una concordancia razonable, hay una buena razón para creer que las estimaciones son fiables. Si, por otro lado, los resultados de estas técnicas de descomposición muestran poca concordancia, se debe realizar más investigación y análisis.

5.6.6. Un ejemplo de estimación basada en el proceso

Para ilustrar el uso de la estimación basada en el proceso, de nuevo consideremos el software de CAD presentado en la Sección 5.6.3. La configuración del sistema y todas las funciones del software permanecen iguales y están indicadas en el ámbito del proyecto.



Si el tiempo lo permite, realice uno mayor descomposición al especificar las tareas; en la Figura 5.5, p. ej., se divide el análisis en sus tareas principales y se estima coda una por separado.

En la tabla de estimación de esfuerzos ya terminada, que se muestra en la Figura *5.5* aparecen las estimaciones de esfuerzo (en personas-mes) para cada actividad de ingeniería del software proporcionada para cada función del software de CAD (abreviada para mayor rapidez). Las actividades de *ingeniería* y de *construcciónlentrega* se subdividen en las tareas principales de ingeniería del software mostradas. Las estimaciones iniciales del esfuerzo se proporcionan para la *comunicación con el cliente, planificación y análisis de riesgos.* Estos se señalan en la fila **Total** en la parte inferior de la tabla. Los totales horizontales y verticales proporcionan una indicación del esfuerzo estimado que se requiere para el análisis, diseño, codificación y pruebas. Se debería señalar que el 53 por 100 de todo el esfuerzo se aplica en las tareas de ingeniería «frontales» (análisis y diseño de los requisitos) indicando la importancia relativa de este trabajo.

Basado en una tarifa laboral de £5.000 por mes, el coste total estimado del proyecto es de £230.000 y el esfuerzo estimado es de 46 personas/mes. A cada actividad de proceso del software o tareas de ingeniería del software se le podría asociar diferentes índices de trabajo y calcularlos por separado.

El esfuerzo total estimado para el software de CAD oscila de un índice bajo de 46 personas-mes (obtenido mediante un enfoque de estimación basado en el proceso) a un alto de 58 personas-mes (obtenido mediante un enfoque de estimación de PF). La estimación media (utilizando los tres enfoques) es de 53 personas-mes. La variación máxima de la estimación media es aproximadamente del 13 por ciento.

¿Qué ocurre cuando la concordancia entre las estimaciones es mala? Para responder a esta pregunta se ha

de reevaluar la información que se ha utilizado para hacer las estimaciones.

Muchas divergencias entre estimaciones se deben a una de dos causas:



No espere que todos los estimaciones coincidan en uno o dos porcentajes. Si lo estimación está dentro de uno banda del 20 por 100, puede reconciliarse en un único valor.

1. No se entiende adecuadamente el ámbito del proyecto o el planificador lo ha mal interpretado.
2. Los datos de productividad usados para técnicas de estimación basados en problemas no son apropiados para la aplicación, están obsoletos (ya no reflejan con precisión la organización de la ingeniería del software), o se han aplicado erróneamente.

El planificador debe determinar la causa de la divergencia y conciliar las estimaciones.

5.7 MODELOS EMPÍRICOS DE ESTIMACIÓN

Un *modelo de estimación* para el software de computadora utiliza fórmulas derivadas empíricamente para predecir el esfuerzo como una función de LDC o PF. Los valores para LDC o PF son estimados utilizando el enfoque descrito en las Secciones *5.6.2* y *5.6.3.* Pero en lugar de utilizar las tablas descritas en esas secciones, los valores resultantes para LDC o PF se unen al modelo de estimación. Los datos empíricos que soportan la mayoría de los modelos de estimación se obtienen de una muestra limitada de proyectos. Por esta razón, el modelo de estimación no es adecuado para todas las clases de software y en todos los entornos de desarrollo. Por consiguiente, dos resultados obtenidos de dichos modelos se deben utilizar con prudencia[13].

5.7.1. La estructura de los modelos de estimación



Un modelo de estimación refleja lo cantidad de proyectos a partir de donde se ha obtenido. Por consiguiente, el modelo es susceptible al dominio.

Un modelo común de estimación se extrae utilizando el análisis de regresión sobre los datos recopilados de proyectos de software anteriores. La estructura global de tales modelos adquiere la forma de [MAT94]:

$$E = A + B \times (ev)^C \qquad (5.2)$$

donde A, B y C son constantes obtenidas empíricamente, E es el esfuerzo en personas-mes, y ev es la variable de estimación (de LDC o PF). Además de la relación señalada en la Ecuación (5.2) la mayoría de los modelos de estimación tienen algún componente de ajuste del proyecto que permite ajustar E por otras características del proyecto (por ejemplo: complejidad del problema, experiencia del personal, entorno de desarrollo). Entre los muchos modelos de estimación orientados a LDC propuestos en la bibliografía se encuentran los siguientes:

$E = 5{,}2 \times (KLDC)^{0{,}91}$	Modelo de Walston-Felix
$E = 5{,}5 + 0{,}73 \times (KLDC)^{1{,}16}$	Modelo de Bailey-Basili
$E = 3{,}2 \times (KLDC)^{1{,}05}$	Modelo simple de Boehm
$E = 5{,}288 \times (KLDC)^{1{,}047}$	Modelo Doty para KLDC > 9

También se han propuesto los modelos orientados a PF. Entre estos modelos se incluyen:

$E = -13{,}39 + 0{,}0545\ PF$	Modelo de Albretch y Gaffney
$E = 60{,}62 \times 7{,}728 \times 10^{-8}\ PF^3$	Modelo de Kemerer
$E = 585{,}7 + 15{,}12\ PF$	Modelo de Matson, Bamett y Mellichamp

Un rápido examen de los modelos listados anteriormente indica que cada uno producirá un resultado diferente[14] para el mismo valor de LDC y PF. La implicación es clara. Los modelos de estimación se deben calibrar para necesidades locales.



Ninguno de estos modelos debería ser aplicado inodecuodomente o su entorno.

[13] En general se debería calibrar un modelo de estimación para las condiciones locales. El modelo se debería ejecutar utilizando los resultados de proyectos terminados. Los datos previstos por el modelo se deberían comparar con los resultados reales, y la eficacia del modelo (para condiciones locales) debería ser evaluada. Si la concordancia no es buena, los coeficientes del modelo se deben volver a calcular utilizando datos locales.

[14] Esta referencia se fundamenta en que estos modelos a menudo se derivan de un número relativamente pequeño de proyectos sólo en unos cuantos dominios de la aplicación.

5.7.2. El modelo COCOMO

Barry Boehm [BOE81], en su libro clásico sobre «economía de la ingeniería del software», introduce una jerarquía de modelos de estimación de software con el nombre de COCOMO, por *Constructive Cost Model* (Modelo Constructivo de Coste). El modelo COCOMO original se ha convertido en uno de los modelos de estimación de coste del software más utilizados y estudiados en la industria. El modelo original ha evolucionado a un modelo de estimación más completo llamado COCOMO II [BOE 96, BOE 00]. **Al** igual que su predecesor, COCOMO 11 es en realidad una jerarquía de modelos de estimación que tratan las áreas siguientes :

Modelo de composición de aplicación. Utilizado durante las primeras etapas de la ingeniería del software, donde el prototipado de las interfaces de usuario, la interacción del sistema y del software, la evaluación del rendimiento, y la evaluación de la madurez de la tecnología son de suma importancia.

Modelo de fase de diseño previo. Utilizado una vez que se han estabilizado los requisitos y que se ha establecido la arquitectura básica del software.

Modelo de fase posterior a la arquitectura. Utilizado durante la construcción del software.

Al igual que todos los modelos de estimación del software, el modelo COCOMO II descrito antes necesita información del tamaño. Dentro de la jerarquía del modelo hay tres opciones de tamaño distintas: puntos objeto, puntos de función, y líneas de código fuente.

El modelo de composición de aplicación COCOMO II utiliza los puntos objeto como se ilustra en los párrafos siguientes. Debería señalarse que también están disponibles otros modelos de estimación más sofisticados (utilizando PF y KLDC) que forman parte del COCOMO II.



Se puede obtener información detallada sobre el COCOMO 11, incluyendo software que puede descargar, en **sunset.usc.edu/COCOMOII/cocomo.html**

Del mismo modo que los puntos de función (Capítulo 4), el *punto objeto* es una medida indirecta de software que se calcula utilizando el total de (1) pantallas (de la interface de usuario), (2) informes, y *(3)* componentes que probablemente se necesiten para construir la aplicación. Cada instancia de objeto (por ejemplo, una pantalla o informe) se clasifica en uno de los tres niveles de complejidad (esto es, básico, intermedio, o avanzado) utilizando los criterios sugeridos por Boehm [BOE96]. En esencia, la complejidad es una función del número y origen de las tablas de datos del cliente y servidor necesario para generar la pantalla o el informe y el número de vistas o secciones presentadas como parte de la pantalla o del informe.

Tipo de objeto	Peso de la Complejidad		
	Básico	Intermedio	Avanzado
Pantalla	1	**2**	**3**
informe	**2**	**5**	a
Componente L3G			**10**

TABLA 5.1. Factores de peso de la complejidad para tipos de objeto [BOE96].

Una vez que se ha determinado la complejidad, se valora el número de pantallas, informes, y componentes de acuerdo con la Tabla 5.1. La cuenta de punto objeto se determina multiplicando el número original de instancias del objeto por el factor de peso de la tabla 5.1 y se suman para obtener la cuenta total de punto objeto. Cuando se va a aplicar el desarrollo basado en componentes o la reutilización de software en general, se estima el porcentaje de reutilización (%reutilización) y se ajusta la cuenta del punto objeto:

PON = (**puntos** objeto) **x** [(100 - %reutilización)/100]

donde PON significa «puntos objeto nuevos».



Para obtener una estimación del esfuerzo basada en el valor PON calculado, se debe calcular la «proporción de productividad». La Tabla 5.2 presenta la proporción de productividad para los diferentes niveles de experiencia del desarrollador **y** de madurez del entorno de desarrollo. Una vez determinada la proporción de productividad, se puede obtener una estimación del esfuerzo del proyecto:

Esfuerzo estimado = PON / PROD

En modelos COCOMO II más avanzados[15], se requiere una variedad de factores de escala, conductores de coste y procedimientos de ajuste. Un estudio completo de estos temas están fuera del ámbito de este libro. El lector interesado debería consultar [BOEOO] o visitar el sitio web COCOMO II.

[15] Como se señaló anteriormente, estos modelos hacen uso de las cuentas de PF y KLDC para la variable tamaño.

Experiencia/capacidad del desarrollador	Muy baja	Baja	Normal	Alta	Muy alta
Madurez/capacidad del entorno	Muy baja	Baja	Normal	Alta	Muy alta
PROD	4	7	13	25	50

TABLA 5.2. Proporciones de productividad para puntos objeto [BOE96].

5.7.3. La ecuación del software

La ecuación del software [PUT92] es un modelo multivariable dinámico que asume una distribución específica del esfuerzo a lo largo de la vida de un proyecto de desarrollo de software. El modelo se ha obtenido a partir de los datos de productividad para unos 4.000 proyectos actuales de software, un modelo de estimación tiene esta forma:

$$E = [LDC \times B^{0,333}/P]^3 \times (1/t^4) \qquad (5.3)$$

donde

E = esfuerzo en personas-mes o personas-año,
t = duración del proyecto en meses o años,
B = «factor especial de destrezas»[16],
P = «parámetro de productividad» que refleja:

- Madurez global del proceso y de las prácticas de gestión.
- La amplitud hasta donde se utilizan correctamente las normas de la ingeniería del software.
- El nivel de los lenguajes de programación utilizados. El estado del entorno del software.
- Las habilidades y la experiencia del equipo del software.
- La complejidad de la aplicación.

Los valores típicos para el desarrollo del software empotrado en tiempo real podrían ser P = 2.000; P = 10.000 para telecomunicaciones y software de sistemas; y P = 28.000 para aplicaciones comerciales de sistemas[17]. El parámetro de productividad se puede extraer para las condiciones locales mediante datos históricos recopilados de esfuerzos de desarrollo pasados.



Se puede obtener información de las herramientas de estimación del coste del software que se han desarrollado a partir de la ecuación del software en **www.qsm.com**

Es importante señalar que la ecuación del software tiene dos parámetros independientes: (1) una estimación del tamaño (en LDC) y (2) una indicación de la duración del proyecto en meses o años.

Para simplificar el proceso de estimación y utilizar una forma más común para su modelo de estimación, Putnam y Myers sugieren un conjunto de ecuaciones obtenidas de la ecuación del software. Un tiempo mínimo de desarrollo se define como:

$$t_{min} = 8,14\ (LDC/P)^{0,43} \text{ en meses para } t_{min} > 6 \text{ meses} \qquad (5.4a)$$

$$E = 180Bt^3 \text{ en personas-mes para } E \geq 20 \text{ personas-mes} \qquad (5.4b)$$

Tenga en cuenta que t en la ecuación (5.4b) se representa en años.

Mediante las ecuaciones (5.4) con P = 12.000 (valor recomendado para software científico) para el software de CAD estudiado anteriormente en este capítulo,

$$t_{min} = 8,14\ (33.200/12.000)^{0,43}$$

$$t_{min} = 12,6 \text{ meses}$$

$$E = 180 \times 0,28 \times (1,05)^3$$

$$E = 58 \text{ personas-mes}$$

Los resultados de la ecuación del software se corresponde favorablemente con las estimaciones desarrolladas en la Sección 5.6. Del mismo modo que el modelo COCOMO expuesto en la sección anterior, la ecuación del software ha evolucionado durante la década pasada. Se puede encontrar un estudio de la versión extendida de este enfoque de estimación en [PUT97].

5.8 LA DECISIÓN DE DESARROLLAR-COMPRAR

En muchas áreas de aplicación del software, a menudo es más rentable adquirir el software de computadora que desarrollarlo. Los gestores de ingeniería del software se enfrentan con una decisión *desarrollar* o *comprar* que se puede complicar aún más con las opciones de adquisición: (1) el software se puede comprar (con licencia) ya desarrollado; (2) se pueden adquirir componentes de software «totalmente experimentado» o «parcialmente experimentado» (Consulte la Sección 5.4.2), y entonces modificarse e integrarse para cumplir las necesidades específicas, o *(3)* el software puede ser construido de forma personalizada por una empresa externa para cumplir las especificaciones del comprador.

[16] B se incrementa lentamente a medida que crecen «la necesidad de integración, pruebas, garantía de calidad, documentación y habilidad de gestión» [PUT92]. Para programas pequeños (KLDC = 5 a 15).B = 0,16. Para programas mayores de 70 KLDC, B = 0,39.

[17] Es importante destacar que el parámetro de productividad puede obtenerse empíricamente de los datos locales de un proyecto.

Los pasos dados en la adquisición del software se definen según el sentido crítico del software que se va a comprar y el coste final. En algunos casos (por ejemplo: software para PC de bajo coste) es menos caro comprar y experimentar que llevar a cabo una larga evaluación de potenciales paquetes de software. Para productos de software más caros, se pueden aplicar las directrices siguientes:

1. Desarrollo de una especificación para la función y rendimiento del software deseado. Definición de las características medibles, siempre que sea posible.



Hoy veces en las que el software proporciono una solución «perfecta» excepto poro algunos aspectos especiales de los que puede prescindir. En muchos casos, ¡merece lo peno vivir sin éstos!

2. Estimación del coste interno de desarrollo y la fecha de entrega.

3a. Selección de tres o cuatro aplicaciones candidatos que cumplan mejor las especificaciones.

3b. Selección de componentes de software reutilizables que ayudarán en la construcción de la aplicación requerida.

4. Desarrollo de una matriz de comparación que presente la comparación una a una de las funciones clave. Alternativamente, realizar el seguimiento de las pruebas de evaluación para comparar el software candidato.

5. Evaluación de cada paquete de software o componente según la calidad de productos anteriores, soporte del vendedor, dirección del producto, reputación, etc.

6. Contacto con otros usuarios de dicho software y petición de opiniones.

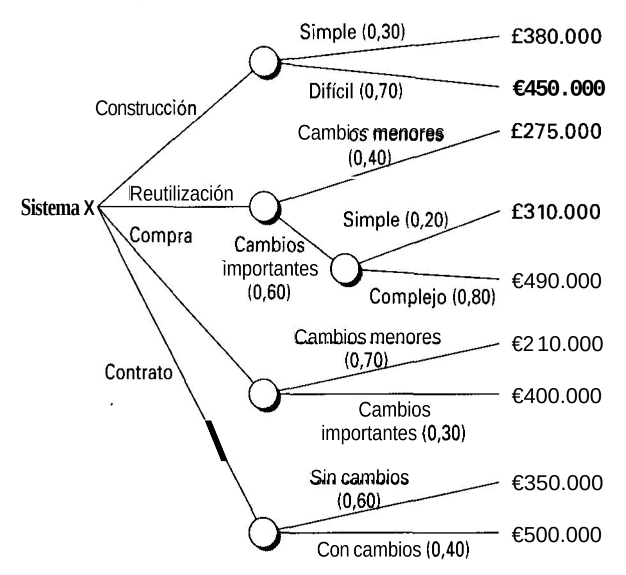


FIGURA 5.6. Árbol de decisión para apoyar la decisión desarrollar-comprar.

En el análisis final, la decisión de desarrollar—comprar se basa en las condiciones siguientes: (1) ¿La fecha de entrega del producto de software será anterior que la del software desarrollado internamente? (2) ¿Será el coste de adquisición junto con el de personalización menor que el coste de desarrollo interno del software? (3) ¿Será el coste del soporte externo (por ejemplo: un contrato de mantenimiento) menor que el coste del soporte interno? Estas condiciones se aplican a cada una de las opciones señaladas anteriormente.

5.8.1. Creación de un árbol de decisiones

Los pasos descritos anteriormente se pueden especificar mediante técnicas estadísticas tales como el *análisis del árbol de decisión* [BOE89]. Por ejemplo, la Figura 5.6 representa un árbol de decisión para un sistema *X* basado en software. En este caso, la organización de la ingeniería del software puede (1) construir el sistema *X* desde el principio; (2) reutilizar los componentes existentes de «experiencia parcial» para construir el sistema; (3) comprar un producto de software disponible y modificarlo para cumplir las necesidades locales; o (4) contratar el desarrollo del software a un vendedor externo.

¿Existe un método sistemático para ordenar las opciones relacionadas con la decisión desarrollar-comprar?

Si se va a construir un sistema desde el principio, existe una probabilidad del 70 por 100 de que el trabajo sea difícil. Mediante las técnicas de estimación estudiadas en este capítulo, el planificador del proyecto estima que un esfuerzo de desarrollo difícil costará £450.000. Un esfuerzo de desarrollo «simple» se estima que cueste £380.000. El valor esperado del coste, calculado a lo largo de la rama del árbol de decisión es:

$$\text{coste esperado} = \sum (\text{probabilidad del camino})_i \times (\text{coste estimado del camino})_i$$

donde *i* es el camino del árbol de decisión. Para el camino construir,

$$\text{coste esperado}_{\text{construcción}} = 0{,}30\ (£380K) + 0{,}70\ (£450K) = £429K$$

También se muestran los otros caminos del árbol de decisión, los costes proyectados para la reutilización, compra y contrato, bajo diferentes circunstancias. Los costes esperados de estas rutas son:

$$\text{coste esperado}_{\text{reutilización}} = 0{,}40\ (£275K) + 0{,}60\ [0{,}20(£310K) + 0{,}80(£490K)] = £382K$$

$$\text{coste esperado}_{\text{compra}} = 0{,}70(£210K) + 0{,}30(£400K) = £267K$$

$$\text{coste esperado}_{\text{,,,,,,}} = 0{,}60(£350K) + 0{,}40(£500K)] = £410K$$

Según la probabilidad y los costes proyectados que se han señalado en la Figura 5.6, el coste más bajo esperado es la opción «compra».



Se puede encontrar un tutorial excelente sobre el análisis del árbol de decisión en **www.demon.co.uk/mindtool/dectree.html**

Sin embargo, es importante señalar que se deben considerar muchos criterios —no sólo el coste— durante el proceso de toma de decisiones. La disponibilidad, la experiencia del desarrollador/vendedor/contratante, la conformidad con los requisitos, la política «local», y la probabilidad de cambios son sólo uno de los pocos criterios que pueden afectar la última decisión para construir, volver a utilizar o contratar.

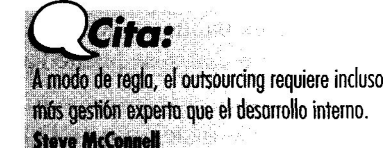
5.8.2. Subcontratación (outsourcing)

Más pronto o más tarde toda compañía que desarrolla software de computadora hace una pregunta fundamental: «¿Existe alguna forma por la que podamos conseguir a bajo precio el software y los sistemas que necesitamos?» La respuesta a esta pregunta no es simple, y las discusiones sobre esta cuestión dan como respuesta una simple palabra: subcontratación (outsourcing).



Se puede encontrar información útil (documentos, enlaces) acerca del outsourcing en **www.outsourcing.com**

El concepto outsourcing es extremadamente simple. Las actividades de ingeniería del software se contratan con un tercero, quien hace el trabajo a bajo coste asegurando una alta calidad. El trabajo de software llevado dentro de la compañía se reduce a una actividad de gestión de contratos.



A modo de regla, el outsourcing requiere incluso más gestión experta que el desarrollo interno.
Steve McConnell

La decisión de contratar fuentes externas puede ser estratégica o táctica. En el nivel estratégico, los gestores tienen en consideración si una parte importante de todo el trabajo del software puede ser contratado a otros. En el nivel táctico, un jefe de proyecto determina si algunas partes o todo el proyecto es aconsejable realizarlo mediante subcontratación.

Con independencia de la amplitud del enfoque, la decisión de elegir outsourcing es a menudo financiera. Un estudio detallado del análisis financiero de la decisión del outsourcing está fuera del ámbito de este libro y se reserva mejor para otros (por ejemplo: [MIN95]). Sin embargo, merece la pena realizar un repaso de los pros y los contras de la decisión.

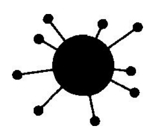
En el lado positivo, los ahorros de coste se pueden lograr reduciendo el número de personas y las instalaciones (por ejemplo: computadoras, infraestructura) que los apoyan. En el lado negativo, una compañía pierde control sobre el software que necesita. Como el software es una tecnología que diferencia sus sistemas, servicios, y productos, una compañía corre el riesgo de poner su destino en manos de un tercero.

5.9 HERRAMIENTAS AUTOMÁTICAS DE ESTIMACIÓN

Las técnicas de descomposición y los modelos empíricos de estimación descritos en las secciones anteriores se pueden implementar con software. Las herramientas automáticas de estimación permiten al planificador estimar costes y esfuerzos, así como llevar a cabo análisis del tipo «qué pasa si» con importantes variables del proyecto, tales como la fecha de entrega o la selección de personal. Aunque existen muchas herramientas automáticas de estimación, todas exhiben las mismas características generales y todas realizan las seis funciones genéricas mostradas a continuación [JON96]:

1. **Dimensionamiento de las entregas del proyecto.** Se estima el «tamaño» de uno o más productos de software. Los productos incluyen la representación externa del software (por ejemplo, pantallas, informes), el software en sí (por ejemplo, KLDC), su funcionalidad (por ejemplo, puntos de función), y la información descriptiva (por ejemplo, documentos).
2. **Selección de las actividades del proyecto.** Se selecciona el marco de trabajo del proceso adecuado (Capítulo 2) y se especifica el conjunto de tareas de ingeniería del software.
3. **Predicción de los niveles de la plantilla.** Se especifica el número de personas disponibles para realizar el trabajo. Esto es muy importante, puesto que la relación entre las personas disponibles y el trabajo (esfuerzo previsto) no es muy lineal.
4. **Predicción del esfuerzo del software.** Las herramientas de estimación utilizan uno o más modelos (por ejemplo, Sección 5.7) que relacionan el tamaño de las entregas del proyecto con el esfuerzo necesario para producirlas.
5. **Predicción del coste del software.** Dado los resultados del paso 4, los costes pueden calcularse asignando proporciones del trabajo a las actividades del proyecto señaladas en el paso 2.
6. **Predicción de la planificación del software.** Cuando se conoce el esfuerzo, los niveles de la plantilla y las actividades del proyecto, se puede realizar un borrador de la planificación asignando el trabajo a través

de actividades de ingeniería del software basadas en modelos recomendados para la distribución del esfuerzo (Capítulo 7).



Herramientas Case.

Cuando se aplican distintas herramientas de estimación a los mismos datos del proyecto, se observa una variación relativamente grande entre los resultados estimados. Pero lo que es más importante, a veces los valores son bastante diferentes de los valores reales. Esto refuerza la idea de que los resultados obtenidos por las herramientas de estimación se deben usar sólo como «punto de partida» para la obtención de estimaciones —no como única fuente para la estimación—.

RESUMEN

El planificador del proyecto de software tiene que estimar tres cosas antes de que comience el proyecto: cuánto durará, cuánto esfuerzo requerirá y cuánta gente estará implicada. Además, el planificador debe predecir los recursos (de hardware y de software) que va a requerir, y el riesgo implicado.

El enunciado del ámbito ayuda a desarrollar estimaciones mediante una o varias de las técnicas siguientes: descomposición, modelos empíricos y herramientas automáticas. Las técnicas de descomposición requieren un esbozo de las principales funciones del software, seguido de estimaciones (1) del número de LDC's, (2) de los valores seleccionados dentro del dominio de la información, (3) del número de personas-mes requeridas para implementar cada función, o (4) del número de personas-mes requeridas para cada actividad de ingeniería del software. Las técnicas empíricas usan expresiones empíricamente obtenidas para el esfuerzo y para el tiempo, con las que se predicen esas magnitudes del proyecto. Las herramientas automáticas implementan un determinado modelo empírico.

Para obtener estimaciones exactas para un proyecto, generalmente se utilizan al menos dos de las tres técnicas referidas anteriormente. Mediante la comparación y la conciliación de las estimaciones obtenidas con las diferentes técnicas, el planificador puede obtener una estimación más exacta. La estimación del proyecto de software nunca será una ciencia exacta, pero la combinación de buenos datos históricos y de técnicas sistemáticas pueden mejorar la precisión de la estimación.

REFERENCIAS

[BEN92] Bennatan, E. M., *Software Project Management: A Practitioner's Approach,* McGraw-Hill, 1992.

[BOE81] Boehm, B., *Software Engineering Economics,* Prentice-Hall, 1981.

[BOE89] Boehm, B., *Risk Management,* IEEE Computer Society Press, 1989.

[BOE96] Boehm, B., «Anchoring the Software Process», *IEE Software,* vol. 13, n.º 4, Julio 1996, pp. 73-82.

[BOE00] Boehm, B., et al., *Software Cost Estimation in COCOMO II,* Prentice Hall, 2000.

[BRO75] Brooks, F., *The Mythical Man-Month,* Addison-Wesley, 1975.

[GAU89] Gause, D. C., y G. M. Weinberg, *Exploring Requirements: Quality Before Design,* Dorset House, 1989.

[HOO91] Hooper, J. W. E., y R. O. Chester, *Software Reuse: Guidelines and Methods,* Plenum Press, 1991.

[JON96] Jones, C., «How Software Estimation Tools Work», *American Programmer,* vol. 9, n.º 7, Julio 1996, pp. 19-27.

[MAH96] Mah, M., *Quantitive Software Management,* Inc., private communication.

[MAT94] Matson, J., B. Barrett y J. Mellichamp, «Software Development Cost Estimation Using Function Points», *IEEE Trans. Software Engineering,* vol. 20, n.º 4, Abril 1994, pp. 275-287.

[MIN95] Minoli, D., *Analizing Outsourcing,* McGraw-Hill, 1995.

[PHI98] Phillips, D., *The Software Project Manager's Handbook,* IEEE Computer Society Press, 1998.

[PUT92] Putnam, L., y W. Myers, *Measures for Excellence,* Yourdon Press, 1992.

[PUT97a] Putnam, L., y W. Myers, «How Solved is the Cost Estimation Problem», *IEEE Software,* Noviembre 1997, pp. 105-107.

[PUT97b] Putnam, L., y W. Myers, *Industrial Strength Software: Effective Management Using Measurement,* IEEE Computer Society Press, 1997.

PROBLEMAS Y PUNTOS A CONSIDERAR

5.1. Suponga que es el gestor de proyectos de una compañía que construye software para productos de consumo. Ha contratado una construcción de software para un sistema de seguridad del hogar. Escriba una especificación del ámbito que describa el software. Asegúrese de que su enunciado del ámbito sea limitado. Si no está familiarizado con sistemas de seguridad del hogar, investigue un poco antes de comenzar a escribir. *Alternativa:* sustituya el sistema de seguridad del hogar por otro problema que le sea de interés.

5.2. La complejidad del proyecto de software se trata brevemente en la Sección 5.1. Desarrolle una lista de las características de software (por ejemplo: operación concurrente, salida gráfica, etc.), que afecte a la complejidad de un proyecto. Dé prioridad a la lista.

5.3. El rendimiento es una consideración importante durante la planificación. Discuta si puede interpretar el rendimiento de otra manera, dependiendo del área de aplicación del software.

5.4. Haga una descomposición de las funciones software para el sistema de seguridad del hogar descrita en el Problema 5.1. Estime el tamaño de cada función en LDC. Asumiendo que su organización produce 450LDC/pm con una tarifa laboral de $7.000 por persona-mes, estime el esfuerzo y el coste requerido para construir el software utilizando la técnica de estimación basada en LDC que se describe en la Sección 5.6.3.

5.5. Utilizando las medidas de punto de función de tres dimensiones que se describe en el Capítulo 4, calcule el número de PF para el software del sistema de seguridad del hogar, y extraiga las estimaciones del esfuerzo y del coste mediante la técnica de estimación basada en PF que se describe en la Sección 5.6.4.

5.6. Utilice el modelo COCOMO II para estimar el esfuerzo necesario para construir software para un simple ATM que produzca 12 pantallas; 10 informes, y que necesite aproximadamente 80 componentes de software. Asuma complejidad «media» y maduración desarrollador/entorno media. Utilice el modelo de composición de aplicación con puntos objeto.

5.7. Utilice la «ecuación del software» para estimar el software del sistema de seguridad del hogar. Suponga que las Ecuaciones (5.4) son aplicables y que P = 8.000.

5.8. Compare las estimaciones de esfuerzo obtenidas de los Problemas 5.5 y 5.7. Desarrolle una estimación simple para el proyecto mediante una estimación de tres puntos. ¿Cuál es la desviación estándar?, y ¿cómo afecta a su grado de certeza sobre la estimación?

5.9. Mediante los resultados obtenidos del Problema 5.8, determine si es razonable esperar que el resultado se pueda construir dentro de los seis meses siguientes y cuántas personas se tendrían que utilizar para terminar el trabajo.

5.10. Desarrolle un modelo de hoja de cálculo que implemente una técnica de estimación o más, descritas en el capítulo. Alternativamente, obtenga uno o más modelos directos para la estimación desde la web.

5.11. *Para un equipo de proyecto:* Desarrolle una herramienta de software que implemente cada una de las técnicas de estimación desarrolladas en este capítulo.

5.12. Parece extraño que las estimaciones de costes y planificaciones temporales se desarrollen durante la planificación del proyecto de software —antes de haber realizado un análisis de requisitos o un diseño detallado—. ¿Por qué cree que se hace así? ¿Existen circunstancias en las que no deba procederse de esta forma?

5.13. Vuelva a calcular los valores esperados señalados en el árbol de decisión de la Figura 5.6 suponiendo que todas las ramas tienen una probabilidad de 50-50. ¿Por qué cambia esto su decisión final?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

La mayoría de los libros de gestión de proyectos de software contienen estudios sobre la estimación de proyectos. Jones *(Estimating Software Costs,* McGraw Hill, 1998) ha escrito **el** tratado más extenso sobre la estimación de proyectos publicado hasta la fecha. Su libro contiene modelos y datos aplicables a la estimación del software en todo el dominio de la aplicación. Roetzheim y Beasley *(Software Project Cost and Schedule Estimating:* Best Practices, Prentice-Hall, 1997) presentan varios modelos útiles y sugieren las directrices paso a paso para generar las estimaciones mejores posibles.

Los libros de Phillips [PHI98], Bennatan *(On Time, Within Budget: Software Project Management Practices and Techniques,* Wiley, 1995), Whitten *(Managing Software Development Projects: Formula for Success,* Wiley, 1995), Wellman *(Software Costing,* Prentice-Hall, 1992), Londeix *(Cost Estimation for Software Development,* Addison-Wesley, 1987) contienen información útil sobre la planificación y la estimación de proyectos de software.

El tratamiento detallado de la estimación del coste de software de Putnam y Myers ([PUT92] y [PUT97b]), y el libro de Boehm sobre la economía de la ingeniería del software ([BOE81] y COCOMO II [BOE00]) describen los modelos de estimación empíricos para la estimación del software. Estos libros proporcionan un análisis detallado de datos que provienen de cientos de proyectos de software. Un libro excelente de DeMarco (*Controlling Software Projects,* Yourdon Press, 1982) proporciona una profunda y valiosa visión de la gestión, medición y estimación de proyectos de software. Sneed *(Software Engineering Management,* Wiley, 1989) y Macro *(Software Engineering: Concepts and Management,* Prentice—Hall, 1990) estudian la estimación del proyecto de software con gran detalle.

La estimación del coste de líneas de código es el enfoque más comúnmente utilizado. Sin embargo, el impacto del paradigma orientado a objetos (consulte la Parte Cuarta) puede invalidar algunos modelos de estimación. Lorenz y Kidd *(Object—Oriented Software Metrics,* Prentice—Hall, 1994) y Cockburn *(Surviving Object—Oriented Projects, Addison–Wesley, 1998)* estudian la estimación para sistemas orientados a objetos.

En Intemet están disponibles una gran variedad de fuentes de información sobre la planificación y estimación del software. Se puede encontrar una lista actualizada con referencias a sitios (páginas) web que son relevantes para la estimación del software en **http://www.pressman5.com.**

CAPÍTULO

6 ANÁLISIS Y GESTIÓN DEL RIESGO

EN su libro sobre análisis y gestión del riesgo, Robert Charette [CHA89] presenta la siguiente definición de riesgo:

En primer lugar, el riesgo afecta a los futuros acontecimientos. El hoy y el ayer están más allá de lo que nos pueda preocupar, pues ya estamos cosechando lo que sembramos previamente con nuestras acciones del pasado. La pregunta es, podemos, por tanto, cambiando nuestras acciones actuales, crear una oportunidad para una situación diferente y, con suerte, mejor para nosotros en el futuro. Esto significa, en segundo lugar, que el riesgo implica cambio, que puede venir dado por cambios de opinión, de acciones, de lugares... [En tercer lugar,] el riesgo implica elección, y la incertidumbre que entraña la elección. Por tanto, el riesgo, como la muerte y los impuestos, es una de las pocas cosas inevitables en la vida.

Cuando se considera el riesgo en el contexto de la ingeniería del software, los tres pilares conceptuales de Charette se hacen continuamente evidentes. El futuro es lo que nos preocupa —¿qué riesgos podrían hacer que nuestro proyecto fracasara?—. El cambio es nuestra preocupación ¿cómo afectarán los cambios en los requisitos del cliente, en las tecnologías de desarrollo, en las computadoras a las que va dirigido el proyecto y a todas las entidades relacionadas con él, al cumplimiento de la planificación temporal y al éxito en general?. Para terminar, debemos enfrentarnos a decisiones —¿qué métodos y herramientas deberíamos emplear, cuánta gente debería estar implicada, cuánta importancia hay que darle a la calidad?—.

VISTAZO RÁPIDO

¿Qué es? El análisis y la gestión del riesgo son una serie de pasos que ayudan al equipo del software a comprender y a gestionar la incertidumbre. Un proyecto de software puede estar lleno de problemas. Un riesgo es un problema potencial —puede ocurrir o no—. Pero sin tener en cuenta el resultado, realmente es una buena idea identificarlo, evaluar su probabilidad de aparición, estimar su impacto, y establecer un plan de contingencia por si ocurre el problema.

¿Quien lo hace? Todos los que estén involucrados en el proceso del software –gestores, ingenieros de software y clientes— participan en el análisis y la gestión del riesgo.

¿Por qué es importante? Pensemos en el lema de los boys scout: «estar preparados». El software es una empresa difícil. Muchas cosas pueden ir mal, y francamente, a menudo van mal. Esta es la razón para estar preparados -comprendiendo los riesgos y tomando las medidas proactivas para evitarlo o gestionarlo— es un elemento clave de una buena gestión de proyectos de software.

¿Cuáles son los pasos? El reconocimiento de que algo puede ir mal es el primer paso, llamado identificación del riesgo. A continuación, cada riesgo es analizado para determinar la probabilidad de que pueda ocurrir y el daño que puede causar si ocurre. Una vez establecida esta información, se priorizan los riesgos, en función de la probabilidad y del impacto. Por último, se desarrolla un plan para gestionar aquellos riesgos con gran probabilidad e impacto.

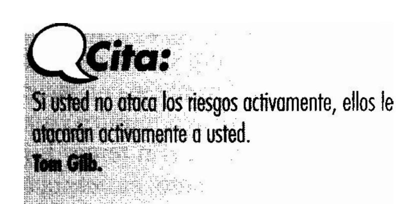
¿Cuál es el producto obtenido? Se realiza un Plan de Reducción, Supervisión y Gestión del Riesgo (RSGR) o un informe de riesgos.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Los riesgos analizados y gestionados deberían derivarse del estudio del personal, del producto, del proceso y del proyecto. El RSGR debe ser revisado mientras el proyecto se realiza para asegurar que los riesgos están siendo controlados hasta la fecha. Los planes de contingencia para la gestión del riesgo deben ser realistas.

Peter Drucker [DRU75] dijo una vez: «Mientras que es inútil intentar eliminar el riesgo y cuestionable el poder minimizarlo, es esencial que los riesgos que se tomen sean los riesgos adecuados». Antes de poder identificar los «riesgos adecuados» a tener en cuenta en un proyecto de software, es importante poder identificar todos los riesgos que sean obvios a jefes de proyecto y profesionales del software.

6.1 ESTRATEGIAS DE RIESGO PROACTIVAS VS. REACTIVAS

Las estrategias se han denominado humorísticamente «Escuela de gestión del riesgo de Indiana Jones» [THO92]. En las películas, Indiana Jones, cuando se enfrentaba a una dificultad insuperable, siempre decía, «¡No te preocupes, pensaré en algo!». Nunca se preocupaba de los problemas hasta que ocurrían, entonces Indy reaccionaba de alguna manera heroica.



Desgraciadamente, el jefe del proyecto de software normalmente no es Indiana Jones y los miembros de su equipo no son sus fiables colaboradores. Sin embargo, la mayoría de los equipos de software confían solamente en las estrategias de riesgo reactivas. En el mejor de los casos, la estrategia reactiva supervisa el proyecto en previsión de posibles riesgos. Los recursos se ponen aparte, en caso de que pudieran convertirse en problemas reales. Pero lo más frecuente es que el equipo de software no haga nada respecto a los riesgos hasta que algo va mal. Después el equipo vuela para corregir el problema rápidamente. Este es el método denominado a menudo «de bomberos». Cuando falla, «la gestión de crisis» [CHA92] entra en acción y el proyecto se encuentra en peligro real.

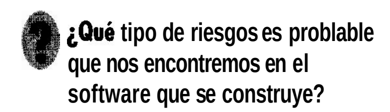
Una estrategia considerablemente más inteligente para el control del riesgo es ser proactivo. La estrategia proactiva empieza mucho antes de que comiencen los trabajos técnicos. Se identifican los riesgos potenciales, se evalúa su probabilidad y su impacto y se establece una prioridad según su importancia. Después, el equipo de Software establece un plan para controlar el riesgo. El primer objetivo es evitar el riesgo, pero como no se pueden evitar todos los riesgos, el equipo trabaja para desarrollar un plan de contingencia que le permita responder de una manera eficaz y controlada. A lo largo de lo que queda de este capítulo, estudiamos la estrategia proactiva para el control de riesgos.

6.2 RIESGO DEL SOFTWARE

Aunque ha habido amplios debates sobre la definición adecuada para riesgo de software, hay acuerdo común en que el riesgo siempre implica dos características [HIG95]:

- *Incertidumbre:* el acontecimiento que caracteriza al riesgo puede o no puede ocurrir; por ejemplo, no hay riesgos de un 100 por 100 de probabilidad[1].
- *Pérdida:* si el riesgo se convierte en una realidad, ocurrirán consecuencias no deseadas o pérdidas.

Cuando se analizan los riesgos es importante cuantificar el nivel de incertidumbre y el grado de pérdidas asociado con cada riesgo. Para hacerlo, se consideran diferentes categorías de riesgos.



Los riesgos del proyecto amenazan al plan del proyecto; es decir, si los riesgos del proyecto se hacen realidad, es probable que la planificación temporal del proyecto se retrase y que los costes aumenten. Los riesgos del proyecto identifican los problemas potenciales de presupuesto, planificación temporal, personal (asignación y organización), recursos, cliente y requisitos y su impacto en un proyecto de software. En el Capítulo 5, la complejidad del proyecto, tamaño y el grado de incertidumbre estructural fueron también definidos como factores (y estimados) de riesgo del proyecto.

Los riesgos técnicos amenazan la calidad y la planificación temporal del software que hay que producir. Si un riesgo técnico se convierte en realidad, la implementación puede llegar a ser difícil o imposible. Los riesgos técnicos identifican problemas potenciales de diseño, implementación, de interfaz, verificación y de mantenimiento. Además, las ambigüedades de especificaciones, incertidumbre técnica, técnicas anticuadas y las «tecnologías punta» son también factores de riesgo. Los riesgos técnicos ocurren porque el problema es más difícil de resolver de lo que pensábamos.

Los *riesgos del negocio* amenazan la viabilidad del software a construir. Los riesgos del negocio a menudo ponen en peligro el proyecto o el producto. Los candidatos para los cinco principales riesgos del negocio son: (1) construir un producto o sistema excelente que no quiere nadie en realidad (riesgo de mercado); (2) construir un producto que no encaja en la estrategia comercial general de la compañía (riesgo estratégico); *(3)* construir un producto que el departamento de ventas no

[1] Un riesgo del 100 por 100 es una limitación del proyecto de software.

sabe cómo vender; (4) perder el apoyo de una gestión experta debido a cambios de enfoque o a cambios de personal (riesgo de dirección); (5) perder presupuesto o personal asignado (riesgos de presupuesto). Es extremadamente importante recalcar que no siempre funciona una categorización tan sencilla. Algunos riesgos son simplemente imposibles de predecir.

Cita:

[Hoy en día,] nadie se permite el lujo de conocer tan bien una tarea que no contenga ninguna sorpresa, y sorpresas significan riesgo.

Stephen Grey.

Otra categorización general de los riesgos ha sido propuesta por Charette [CHA89]. Los *riesgos conocidos* son todos aquellos que se pueden descubrir después de una cuidadosa evaluación del plan del proyecto, del entorno técnico y comercial en el que se desarrolla el proyecto **y** otras fuentes de información fiables (por ejemplo: fechas de entrega poco realistas, falta de especificación de requisitos o de ámbito del software, o un entorno pobre de desarrollo). Los *riesgos predecibles* se extrapolan de la experiencia en proyectos anteriores (por ejemplo: cambio de personal, mala comunicación con el cliente, disminución del esfuerzo del personal a medida que atienden peticiones de mantenimiento). Los *riesgos impredecibles* son el comodín de la baraja. Pueden ocurrir, pero son extremadamente difíciles de identificar por adelantado.

6.3 IDENTIFICACIÓN DEL RIESGO

La *identificación del riesgo* es un intento sistemático para especificar las amenazas al plan del proyecto (estimaciones, planificación temporal, carga de recursos, etc.). Identificando los riesgos conocidos y predecibles, el gestor del proyecto da un paso adelante para evitarlos cuando sea posible y controlarlos cuando sea necesario.

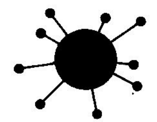


*Aunque los riesgos genéricos son **importantes a** tener en cuenta, **habitualmente** los riesgos específicos del producto provocan lo **mayoría** de los dolores de cabeza. Esté **convencido al** invertir el tiempo en identificar tantos riesgos específicos de/ producto como seo posible.*

Existen dos tipos diferenciados de riesgos para cada categoría presentada en la Sección 6.2: riesgos genéricos y riesgos específicos del producto. *Los riesgos genéricos* son una amenaza potencial para todos los proyectos de software. *Los riesgos específicos de producto* sólo los pueden identificar los que tienen una clara visión de la tecnología, el personal y el entorno específico del proyecto en cuestión. Para identificar los riesgos específicos del producto, se examinan el plan del proyecto **y** la declaración del ámbito del software y se desarrolla una respuesta a la siguiente pregunta: «¿Qué características especiales de este producto pueden estar amenazadas por nuestro plan del proyecto?»

Un método para identificar riesgos es crear una *lista de comprobación de elementos de riesgo*. La lista de comprobación se puede utilizar para identificar riesgos **y** se centra en un subconjunto de riesgos conocidos **y** predecibles en las siguientes subcategorías genéricas:

- *Tamaño del producto:* riesgos asociados con el tamaño general del software a construir o a modificar.
- *Impacto en el negocio:* riesgos asociados con las limitaciones impuestas por la gestión o por el mercado.
- *Características del cliente:* riesgos asociados con la sofisticación del cliente y la habilidad del desarrollador para comunicarse con el cliente en los momentos oportunos.
- *Definición del proceso:* riesgos asociados con el grado de definición del proceso del software y su seguimiento por la organización de desarrollo.
- *Entorno de desarrollo:* riesgos asociados con la disponibilidad y calidad de las herramientas que se van a emplear en la construcción del producto.
- *Tecnología a construir:* riesgos asociados con la complejidad del sistema a construir y la tecnología punta que contiene el sistema.
- *Tamaño y experiencia de la plantilla:* riesgos asociados con la experiencia técnica y de proyectos de los ingenieros del software que van a realizar el trabajo.



lista de comprobación de elementos del riesgo.

La lista de comprobación de elementos de riesgo puede organizarse de diferentes maneras. Se pueden responder a cuestiones relevantes de cada uno de los temas apuntados anteriormente para cada proyecto de software. Las respuestas a estas preguntas permiten al planificador del proyecto estimar el impacto del riesgo. Un formato diferente de lista de comprobación de elementos de riesgo contiene simplemente las características relevantes para cada subcategoría genérica. Finalmente, se lista un conjunto de «componentes y controladores del riesgo» [AFC88] junto con sus probabilidades

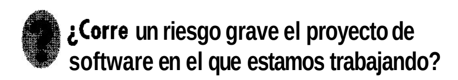
de aparición. Los controladores del rendimiento, el soporte, el coste y la planificación temporal del proyecto se estudian como respuesta a preguntas posteriores.

Se ha propuesto un gran numero de listas de comprobación para el riesgo del proyecto de software en los textos (por ejemplo, [SEI93], [KAR96]). Estos proporcionan una visión útil de riesgos genéricos para proyectos de software y deberían utilizarse cada vez que se realice el análisis y la gestión del riesgo. Sin embargo, se puede utilizar una lista de preguntas relativamente pequeña para proporcionar una indicación preliminar de cuando un proyecto es «arriesgado».



6.3.1. Evaluación Global del Riesgo del Proyecto

Las siguientes preguntas provienen de los datos del riesgo obtenidos mediante las encuestas realizadas a gestores de proyectos de software expertos de diferentes partes del mundo [KEI98]. Las preguntas están ordenadas por su importancia relativa para el éxito de un proyecto.



1. ¿Se han entregado los gestores del software y clientes formalmente para dar soporte al proyecto?
2. ¿Están completamente entusiasmados los usuarios finales con el proyecto y con el sistema/producto a construir?
3. ¿Han comprendido el equipo de ingenieros de software y los clientes todos los requisitos?
4. ¿Han estado los clientes involucrados por completo en la definición de los requisitos?
5. ¿Tienen los usuarios finales expectativas realistas?
6. ¿Es estable el ámbito del proyecto?

COMPONENTES / CATEGORÍA		RENDIMIENTO	SOPORTE	COSTE	PLANIFICACIÓN TEMPORAL
CATASTRÓFICA	1	Dejar de cumplir los requisitos provocaría el fallo de la misión		Malos resultados en un aumento de costes y retrasos de la planificación temporal con gastos que superan las £500.00(	
	2	Degradación significativa para noalcanzar el rendimiento técnico	El software no responde o no admite soporte	Recortes financieros significativos, presupuestos excedidos	Fecha de entrega inalcanzable
CRITICA	1	Dejar de cumplir los requisitos degradaría el rendimiento del sistema hasta donde el éxito de la misión es cuestionable		Malos resultados en retrasos operativos y/o aumento de coste con un valor esperado de €100.000a f500.000	
	2	Alguna reducción en el rendimiento técnico	Pequeños retrasos en modificaciones de software	Algunos recortes de los recursos financieros, posibles excesos del presupuesto	Posibles retrasos en la fecha de entrega
MARGINAL	1	Dejar de cumplir los requisitos provocaría la degradación de la misión secundaria		Los costes, impactos y/o retrasosrecuperables de la planificación temporal con un valor estimado de fl.000 a fl00.000	
	2	De mínima a pequeña reducción en el rendimiento técnico	El soporte del software responde	Recursos financieros suficientes	Planificación temporal realista, alcanzable
DESPRECIABLE	1	Dejar de cumplir los requisitos provocaría inconvenientes o impactos no operativos		Los errores provocan impactos mínimos en el coste y/o planificación temporal con un valor esperado de menos de f1.000	
	2	No hay reducción en el rendimiento técnico	Software fácil de dar soporte	Posible superávit de presupuesto	Fecha de entrega fácilmente alcanzable

Nota : (1) Posibles consecuencias de errores o defectos del software no detectados.
(2) Posibles consecuencias si el resultado deseado no se consigue.

FIGURA 6.2. Evaluación del impacto [BOE89].

7. ¿Tiene el ingeniero de software el conjunto adecuado de habilidades?
8. ¿Son estables los requisitos del proyecto?
9. ¿Tiene experiencia el equipo del proyecto con la tecnología a implementar?
10. ¿Es adecuado el número de personas del equipo del proyecto para realizar el trabajo?
11. ¿Están de acuerdo todos los clientes/usuarios en la importancia del proyecto y en los requisitos del sistema/producto a construir?



«Un radar del riesgo)) es una base de datos de gestión de riesgo que ayuda a los gestores del proyecto a identificar, priorizar y comunicar los riesgos del proyecto. Se puede encontrar en **www.spmn.com/rsktrkr.html**

6.3.2. Componentes y controladores del riesgo

Las Fuerzas Aéreas de Estados Unidos [AFC88] han redactado un documento que contiene excelentes directrices para identificar riesgos del software y evitarlos. El enfoque de las Fuerzas Aéreas requiere que el gestor del proyecto identifique los controladores del riesgo que afectan a los componentes de riesgo del software —rendimiento, coste, soporte y planificación temporal—. En el contexto de este estudio, los componentes de riesgo se definen de la siguiente manera:

- *riesgo de rendimiento:* el grado de incertidumbre con el que el producto encontrará sus requisitos y se adecue para su empleo pretendido;
- *riesgo de coste:* el grado de incertidumbre que mantendrá el presupuesto del proyecto;
- *riesgo de soporte:* el grado de incertidumbre de la facilidad del software para corregirse, adaptarse y ser mejorado;
- *riesgo de la planificación temporal:* el grado de incertidumbre con que se podrá mantener la planificación temporal y de que el producto se entregue a tiempo.

El impacto de cada controlador del riesgo en el componente de riesgo se divide en cuatro categorías de impacto —despreciable, marginal, crítico y catastrófico—. Como muestra la Figura 6.1 [BOE89], se describe una caracterización de las consecuencias potenciales de errores (filas etiquetadas con 1) o la imposibilidad de conseguir el producto deseado (filas etiquetadas con 2). La categoría de impacto es elegida basándose en la caracterización que mejor encaja con la descripción de la tabla.

Riesgos	ategoría	robabilidad	mpacto	RSGR
La estimación del tamaño puedi ser significativamente baja	PS	60%	2	
Mayor número de usuarios de los previstos	PS	30%	3	
Menos reutilización de la prevista	PS	70%	2	
Los usuarios finales se resisten al sistema	BU	40%	3	
La fecha de entrega estará muy ajustada	BU	50%	2	
Se perderán los presupuestos	CU	40%	1	
El cliente cambiará los requisitos	PS	80%	2	
La tecnología no alcanzará las espectativas	TE	30%	1	
Falta de formación en las herramientas	DE	80%	3	
Personal sin experiencia	ST	30%	2	
Habrá muchos cambios de personal	ST	60%	2	

Valores de impacto :
1 - catastrófico   2 - crítico
3 - marginal   4 - despreciable

FIGURA 6.2. Ejemplo de una tabla de riesgo antes de la clasificación.

6.4 PROYECCIÓN DEL RIESGO

La proyección del riesgo, también denominada *estimación del riesgo,* intenta medir cada riesgo de dos maneras —la probabilidad de que el riesgo sea real y las consecuencias de los problemas asociados con el riesgo, si ocurriera—. El jefe del proyecto, junto con otros gestores y personal técnico, realiza cuatro actividades de proyección del riesgo [1]: (1) establecer una escala que refleje la probabilidad percibida del riesgo; (2) definir las consecuencias del riesgo; **(3)** estimar el impacto del riesgo en el proyecto **y** en el producto, y **(4)** apuntar la exactitud general de la proyección del riesgo de manera que no haya confusiones.

6.4.1. Desarrollo de una tabla de riesgo

Una tabla de riesgo le proporciona al jefe del proyecto una sencilla técnica para la proyección del riesgo[2]. En la Figura 6.2 se ilustra una tabla de riesgo como ejemplo.

[2] La tabla de riesgo debería implementarse como un modelo de hoja de cálculo. Esto permite un fácil manejo y ordenación de las entradas.

Un equipo de proyecto empieza por listar todos los riesgos (no importa lo remotos que sean) en la primera columna de la tabla. Se puede hacer con la ayuda de la lista de comprobación de elementos de riesgo presentada en la Sección 6.3. Cada riesgo es categorizado en la segunda columna (por ejemplo: **PS** implica un riesgo del tamaño del proyecto, BU implica un riesgo de negocio). La probabilidad de aparición de cada riesgo se introduce en la siguiente columna de la tabla. El valor de la probabilidad de cada riesgo puede estimarse por cada miembro del equipo individualmente. Se sondea a los miembros del equipo individualmente de un modo rotativo (round-robin) hasta que comience a converger su evaluación sobre la probabilidad del riesgo.

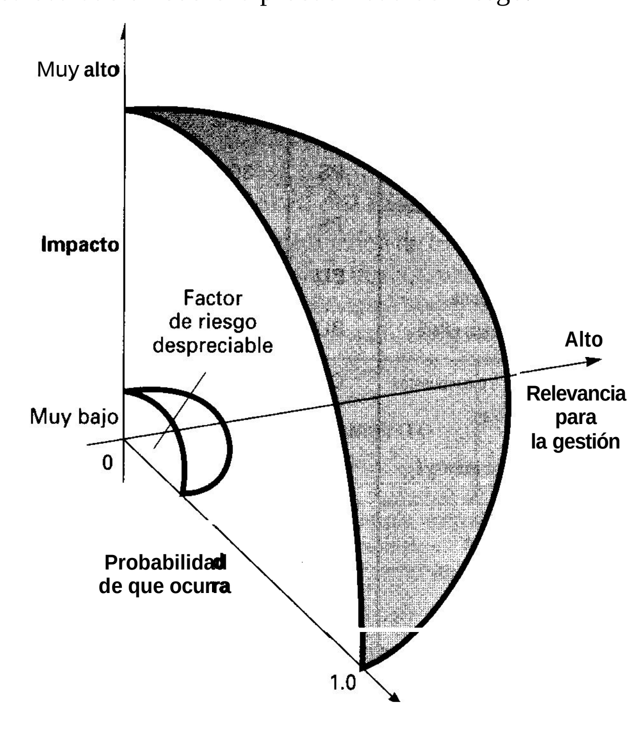


FIGURA 6.3. Riesgos y relevancia para la gestión.

A continuación se valora el impacto de cada riesgo. Cada componente de riesgo se valora usando la caracterización presentada en la Figura 6.1, y se determina una categoría de impacto. Las categorías para cada uno de los cuatro componentes de riesgo —rendimiento, soporte, coste y planificación temporal— son promediados[3] para determinar un valor general de impacto.



Piense seriamente en el software que está construyendo y pregúntese a sí mismo «¿Qué puede ir mal?» Cree su propio listo y consulte a otros miembros del equipo de software para hacer lo mismo.

Una vez que se han completado las cuatro primeras columnas de la tabla de riesgo, la tabla es ordenada por probabilidad y por impacto. Los riesgos de alta probabilidad y de alto impacto pasan a lo alto de la tabla, y los riesgos de baja probabilidad caen a la parte de abajo. Esto consigue una priorización del riesgo de primer orden.

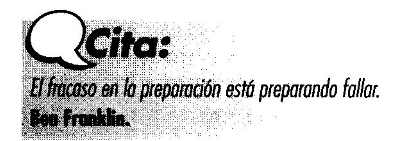
El jefe del proyecto estudia la tabla ordenada resultante y define una línea de corte. *La Eínea de corte* (dibujada horizontalmente desde un punto en la tabla) implica que sólo a los riesgos que quedan por encima de la línea se les prestará atención en adelante. Los riesgos que caen por debajo de la línea son reevaluados para conseguir una priorización de segundo orden. Como muestra la Figura 6.3, el impacto del riesgo y la probabilidad tienen diferente influencia en la gestión. Un factor de riesgo que tenga un gran impacto pero muy poca probabilidad de que ocurra no debería absorber una cantidad significativa de tiempo de gestión. Sin embargo, los riesgos de gran impacto con una probabilidad moderada a alta y los riesgos de poco impacto pero de gran probabilidad deberían tenerse en cuenta en los procedimientos de análisis de riesgos que se estudian a continuación.



CLAVE

la tabla de riesgos está ordenada por probabilidad y por el impacto para asignar una prioridad a los riesgos.

Todos los riesgos que se encuentran por encima de la línea de corte deben ser considerados. La columna etiquetada RSGR* contiene una referencia que apunta hacia *un Plan de reducción, supervisión y gestión* ***del*** *riesgo, o* alternativamente, a un informe del riesgo desarrollado para todos los riesgos que se encuentran por encima de la línea de corte. El plan RSGR y el informe del riesgo se estudian en la Sección 6.5.



La probabilidad de riesgo puede determinarse haciendo estimaciones individuales y desarrollando después un único valor de consenso. Aunque este enfoque es factible, se han desarrollado técnicas más sofisticadas para determinar la probabilidad de riesgo [AFC88]. Los controladores de riesgo pueden valorarse en una escala de probabilidad cualitativa que tiene los siguientes valores: imposible, improbable, probable y frecuente. Después puede asociarse una probabilidad matemática con

3 Puede usarse una media ponderada si un componente de riesgo tiene más influencia en el proyecto.

* En inglés RMMM (Risk Mitigation, Monitoring and Management).

cada valor cualitativo (por ejemplo: una probabilidad del 0.7 al 1.0 implica un riesgo muy probable).

6.4.2. Evaluación del impacto del riesgo

Tres factores afectan a las consecuencias probables de un riesgo si ocurre: su naturaleza, su alcance y cuando ocurre. La naturaleza del riesgo indica los problemas probables que aparecerán si ocurre. Por ejemplo, una interfaz externa mal definida para el hardware del cliente (un riesgo técnico) impedirá un diseño y pruebas tempranas y probablemente lleve a problemas de integración más adelante en el proyecto. El alcance de un riesgo combina la severidad (¿cómo de serio es el problema?) con su distribución general (¿qué proporción del proyecto se verá afectado y cuantos clientes se verán perjudicados?). Finalmente, la temporización de un riesgo considera cuándo y por cuánto tiempo se dejará sentir el impacto. En la mayoría de los casos, un jefe de proyecto prefiere las «malas noticias» cuanto antes, pero en algunos casos, cuanto más tarden, mejor.

Volviendo una vez más al enfoque del análisis de riesgo propuesto por las Fuerzas Aéreas de Estados Unidos [AFC88], se recomiendan los siguientes pasos para determinar las consecuencias generales de un riesgo:

1. Determinar la probabilidad media de que ocurra un valor para cada componente de riesgo.
2. Empleando la Figura 6.1, determinar el impacto de cada componente basándose en los criterios mostrados.
3. Completar la tabla de riesgo y analizar los resultados como se describe en las secciones precedentes.



La exposición al riesgo en general, *ER*, se determina utilizando la siguiente relación [HAL98] :

$$ER = P \times C$$

donde *P* es la probabilidad de que ocurra un riesgo, y *C* es el coste del proyecto si el riesgo ocurriera.

Por ejemplo, supongamos que el equipo del proyecto define un riesgo para el proyecto de la siguiente manera:

Identificación del riesgo : Solo el 70 por 100 de los componentes del software planificados para reutilizarlos pueden, de hecho, integrarse en la aplicación. La funcionalidad restante tendrá que ser desarrollada de un modo personalizado.

Probabilidad del riesgo : 80 por 100 (probable).

Impacto del riesgo : 60 componentes de software reutilizables fueron planificados. Si solo el 70 por 100 pueden usarse, 18 componentes tendrán que desarrollarse improvisadamente (además de otro software personalizado que ha sido planificado para su desarrollo). Puesto que la media por componente es de 100 LDC y los datos locales indican que el coste de la ingeniería del software para cada LDC es de £14,00; el coste global (impacto) para el desarrollo de componentes sería 18 x 100 x 14 = 225.200

Exposición al riesgo : ER = 0,80 x 25.200 - 220.200

La exposición al riesgo se puede calcular para cada riesgo en la tabla de riesgos, una vez que se ha hecho una estimación del coste del riesgo. La exposición al riesgo total para todos los riesgos (sobre la línea de corte en la tabla de riesgos) puede proporcionar un significado para ajustar el coste final estimado para un proyecto. También puede ser usado para predecir el incremento probable de recursos de plantilla necesarios para varios puntos durante la planificación del proyecto.

La proyección del riesgo y las técnicas de análisis descritas en las Secciones 6.4.1 y 6.4.2 se aplican reiteradamente a medida que progresa el proyecto de software. El equipo del proyecto debería volver a la tabla de riesgos a intervalos regulares, volver a evaluar cada riesgo para determinar qué nuevas circunstancias hayan podido cambiar su impacto o probabilidad. Como consecuencia de esta actividad, puede ser necesario añadir nuevos riesgos a la tabla, quitar algunos que ya no sean relevantes y cambiar la posición relativa de otros.



*Compare lo ER poro todos los riesgos con el coste estimado del proyecto. Si lo ER es superior **al 50 por 100** del coste del proyecto, se deberú evaluar lo **viabilidad** del proyecta.*

6.4.3. Evaluación del riesgo

En este punto del proceso de gestión del riesgo, hemos establecido un conjunto de temas de la forma [CHA89]:

$$[r_i, l_i, x_i]$$

donde r_i es el riesgo, l_i es la probabilidad del riesgo y x_i es el impacto del riesgo. Durante la evaluación del riesgo, se sigue examinando la exactitud de las estimaciones que fueron hechas durante la proyección del riesgo, se intenta dar prioridades a los riesgos que no se habían cubierto y se empieza a pensar las maneras de controlar y/o impedir los riesgos que sea más probable que aparezcan.

Para que sea útil la evaluación, se debe definir un nivel de *referencia* de riesgo [CHA89]. Para la mayoría de los proyectos, los componentes de riesgo estudiados anteriormente — rendimiento, coste, soporte y planificación temporal — también representan niveles de referencia de riesgos. Es decir, hay un nivel para la degradación del rendimiento, exceso de coste, dificultades de soporte o retrasos de la planificación temporal (o cualquier combinación de los cuatro) que provoquen que se termine el proyecto. Si una combinación de riesgos crea problemas de manera que uno o más de estos niveles de referencia

se excedan, se parará el trabajo. En el contexto del análisis de riesgos del software, un nivel de referencia de riesgo tiene un solo punto, denominado punto de referencia o punto de ruptura, en el que la decisión de seguir con el proyecto o dejarlo (los problemas son demasiado graves) son igualmente aceptables. La Figura 6.4 representa esta situación gráficamente.

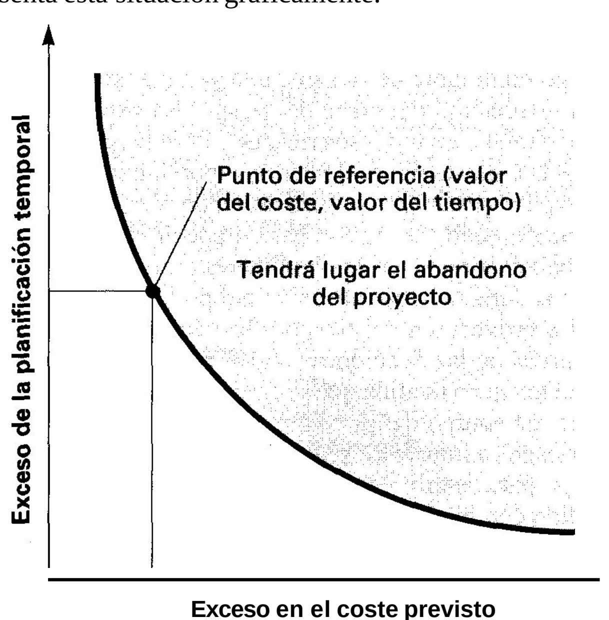


FIGURA 6.4. Nivel de referencia de riesgo.

PUNTO CLAVE

El nivel de referencia del riesgo establece su tolerancia para el sufrimiento. Una vez que la exposición al riesgo supera el nivel de referencia, el proyecto puede ser terminodo.

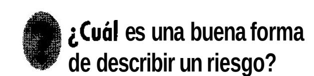
En realidad, el nivel de referencia puede raramente representarse como una línea nítida en el gráfico. En la mayoría de los casos es una región en la que hay áreas de incertidumbre, es decir, intentar predecir una decisión de gestión basándose en la combinación de valores de referencia es a menudo imposible. Por tanto, durante la evaluación del riesgo, se realizan los siguientes pasos:

1. Definir los niveles de referencia de riesgo para el proyecto;
2. Intentar desarrollar una relación entre cada (r_i, l_i, x_i) y cada uno de los niveles de referencia;
3. Predecir el conjunto de puntos de referencia que definan la región de abandono, limitado por una curva o áreas de incertidumbre;
4. Intentar predecir como afectarán las combinaciones compuestas de riesgos a un nivel de referencia.

Es mejor dejar un estudio más detallado para libros especializados en el análisis de riesgos (por ejemplo: [CHA89], [ROW88]).

6.5 REFINAMIENTO DEL RIESGO

Durante las primeras etapas de la planificación del proyecto, un riesgo puede ser declarado de un modo muy general. Con el paso del tiempo y con el aprendizaje sobre el proyecto y sobre el riesgo, es posible refinar el riesgo en un conjunto de riesgos más detallados, cada uno algo más fácil de reducir, supervisar y gestionar.



Una forma de hacer esto es presentar el riesgo de la forma ***condición-transición-consecuencia*** (CTC) [GLU94]. Es decir, el riesgo se presenta de la siguiente forma:

Dada esta <condición> entonces existe preocupación por (posiblemente) <consecuencia>.

Utilizando el formato CTC para volver a utilizar el riesgo presentado en la Sección 6.4.2, podemos escribir:

Dado que todos los componentes reutilizables del software deben ajustarse a los estándares específicos del diseño y que algunos no lo hacen, es entonces preocupante que (posiblemente) solo el 70 por 100 de los módulos planificados para reutilizar puedan realmente integrarse en el sistema que se está construyendo, teniendo como resultado la necesidad de que el ingeniero tenga que construir el 30 por 100 de los componentes restantes.

La condición general que acabamos de destacar puede ser refinada de la siguiente manera:

Subcondición 1: Ciertos componentes reutilizables fueron desarrollados por terceras personas sin el conocimiento de los estándares internos de diseño.

Subcondición 2: El estándar de diseño para interfaces de componentes no ha sido asentado y puede no ajustarse a ciertos componentes reutilizables existentes.

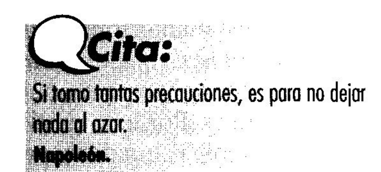
Subcondición 3: Ciertos componentes reutilizables han sido implementados en un lenguaje no soportado por el entorno para el que el sistema ha sido construido.

Las consecuencias relacionadas con estas subcondiciones refinadas siguen siendo las mismas (por ejemplo, el 30 por 100 de los componentes del software deben ser construidos de un modo personalizado), pero el refinamiento ayuda a aislar los riesgos señalados y puede conducir a un análisis y respuesta más sencilla.

6.6 REDUCCIÓN, SUPERVISIÓN Y GESTIÓN DEL RIESGO

Todas las actividades de análisis de riesgo presentadas hasta ahora tienen un solo objetivo —ayudar al equipo del proyecto a desarrollar una estrategia para tratar los riesgos—. Una estrategia eficaz debe considerar tres aspectos:

- Evitar el riesgo.
- Supervisar el riesgo, y
- Gestionar el riesgo y planes de contingencia.



Si un equipo de software adopta un enfoque proactivo frente al riesgo, evitarlo es siempre la mejor estrategia. Esto se consigue desarrollando un plan de *reducción del riesgo*. Por ejemplo, asuma que se ha detectado mucha movilidad de la plantilla como un riesgo del proyecto, r_i. Basándose en casos anteriores y en la intuición de gestión, la probabilidad, l_i, de mucha movilidad se estima en un 0,70 (70 por 100, bastante alto) y el impacto, x_i, está previsto en el nivel 2. Esto es, **un** gran cambio puede tener un impacto crítico en el coste y planificación temporal del proyecto.

Para reducir el riesgo, la gestión del proyecto debe desarrollar una estrategia para reducir la movilidad. Entre los pasos que se pueden tomar:

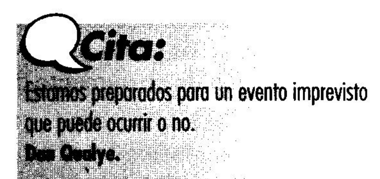
- Reunirse con la plantilla actual y determinar las causas de la movilidad (por ejemplo: malas condiciones de trabajo, salarios bajos, mercado laboral competitivo).
- Actuar para reducir esas causas que estén al alcance de nuestro control antes de que comience el proyecto.
- Una vez que comienza el proyecto, asumir que habrá movilidad y desarrollar técnicas que aseguren la continuidad cuando se vaya la gente.
- Organizar los equipos del proyecto de manera que la información sobre cada actividad de desarrollo esté ampliamente dispersa.
- Definir estándares de documentación y establecer mecanismos para estar seguro de que los documentos se cumplimentan puntualmente.



Se puede obtener uno excelente CPF (cuestiones que se preguntan frecuentemente) en **www.sei.cmu.edu/organization/programs/sepm/risk/risk.faq.html**

A medida que progresa el proyecto, comienzan las actividades de supervisión del riesgo. El jefe del proyecto supervisa factores que pueden proporcionar una indicación sobre si el riesgo se está haciendo más o menos probable. En el caso de gran movilidad del personal, se pueden supervisar los siguientes factores:

- Actitud general de los miembros del equipo basándose en las presiones del proyecto;
- Grado de compenetración del equipo;
- Relaciones interpersonales entre los miembros del equipo;
- Problemas potenciales con compensaciones y beneficios;
- Disponibilidad de empleo dentro y fuera de la compañía.



Además de supervisar los factores apuntados anteriormente, el jefe del proyecto debería supervisar también la efectividad de los pasos de reducción del riesgo. Por ejemplo, un paso de reducción de riesgo apuntado anteriormente instaba a la definición de «estándares de documentación y mecanismos para asegurarse de que los documentos se cumplimenten puntualmente». Éste es un mecanismo para asegurarse la continuidad, en caso de que un individuo crítico abandone el proyecto. El jefe del proyecto debería comprobar los documentos cuidadosamente para asegurarse de que son válidos y de que cada uno contiene la información necesaria en caso de que un miembro nuevo se viera obligado a unirse al equipo de software a mitad del proyecto.

La gestión del riesgo y *los planes de contingencia* asumen que los esfuerzos de reducción han fracasado y que el riesgo se ha convertido en una realidad. Continuando con el ejemplo, el proyecto va muy retrasado y un número de personas anuncia que se va. Si se ha seguido la estrategia de reducción, tendremos copias de seguridad disponibles, la información está documentada y el conocimiento del proyecto se ha dispersado por todo el equipo. Además, el jefe del proyecto puede temporalmente volver a reasignar los recursos (y reajustar la planificación temporal del proyecto) desde **las** funciones que tienen todo su personal, permitiendo a los recién llegados que deben unirse al equipo que vayan «cogiendo el ritmo». **A** los individuos que se van se les pide que dejen lo que estén haciendo y dediquen sus últimas semanas a «transferir sus conocimientos». Esto podría incluir la adquisición de conocimientos por medio de vídeos, el desarrollo de «documentos con comentarios» y/o reuniones con otros miembros del equipo que permanezcan en el proyecto.



Si el valor para un riesgo específico es menor que el coste de la reducción del riesgo, no trate de reducir el riesgo pero continúe supervisándolo.

Es importante advertir que los pasos RSGR provocan aumentos del coste del proyecto. Por ejemplo, emplear tiempo en conseguir «un reserva» de cada técnico crítico cuesta dinero. Parte de la gestión de riesgos es evaluar cuando los beneficios obtenidos por los pasos RSGR superan los costes asociados con su implementación. En esencia, quien planifique el proyecto realiza el clásico análisis coste/beneficio. Si los procedimientos para evitar el riesgo de gran movilidad aumentan el coste y duración del proyecto aproximadamente en un 15 por 100, pero el factor coste principal es la «copia de seguridad», el gestor puede decidir no implementar este paso. Por otra parte si los pasos para evitar el riesgo llevan a una proyección de un aumento de costes del 5 por 100 y de la duración en un 3 por 100, la gestión probablemente lo haga.

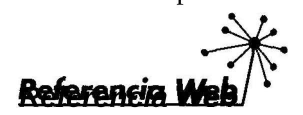
Para un proyecto grande se pueden identificar unos 30 ó 40 riesgos. Si se pueden identificar entre tres y siete pasos de gestión de riesgo para cada uno, ¡la gestión del riesgo puede convertirse en un proyecto por **sí** misma! Por este motivo, adaptamos la regla de Pareto 80/20 al riesgo del software. La experiencia dice que el 80 por 100 del riesgo total de un proyecto (por ejemplo: el 80 por 100 de la probabilidad de fracaso del proyecto) se debe solamente al 20 por 100 de los riesgos identificados. El trabajo realizado durante procesos de análisis de riesgo anteriores ayudará al jefe de proyecto a determinar **qué** riesgos pertenecen a ese 20 por 100 (por ejemplo, riesgos que conducen a una exposición al riesgo alta). Por este motivo, algunos de los riesgos identificados, valorados y previstos pueden no pasar por el plan RSGR —no pertenecen al 20 por 100 crítico— (los riesgos con la mayor prioridad del proyecto).

6.7 RIESGOS Y PELIGROS PARA LA SEGURIDAD

El riesgo no se limita al proyecto de software solamente. Pueden aparecer riesgos después de haber desarrollado con éxito el software y de haberlo entregado al cliente. Estos riesgos están típicamente asociados con las consecuencias del fallo del software una vez en el mercado.

En los comienzos de la informática, había un rechazo al uso de las computadoras (y del software) para el control de procesos críticos de seguridad como por ejemplo reactores nucleares, control de vuelos de aviones, sistemas de armamento y grandes procesos industriales. Aunque la probabilidad de fallo de un sistema de alta ingeniería es pequeña, un defecto no detectado en un sistema de control y supervisión basados en computadora podría provocar unas pérdidas económicas enormes o, peor, daños físicos significativos o pérdida de vidas humanas. Pero el coste y beneficios funcionales del control y supervisión basados en computadora a menudo superan al riesgo. Hoy en día, se emplean regularmente hardware y software para el control de sistemas de seguridad crítica.

Cuando se emplea software como parte de un sistema de control, la complejidad puede aumentar del orden de una magnitud o más. Defectos sutiles de diseño inducidos por error humano —algo que puede descubrirse y eliminarse con controles convencionales basados en hardware— se convierten en mucho más difíciles de descubrir cuando se emplea software.



Se puede encontrar una gran base de datos con todos las entradas del Foro ACM sobre riesgos en **catless.ncl.ac.uk/Risks/search.html**

Hoja de información de riesgo

Id. Riesgo: P02-4-32	fecha: 5/9/02	Probabilidad: 80%	Impacto: alto

Descripción :
Sólo el 70 por 100 de los componentes del software planificados para reutilizar pueden, de hecho, integrarse en la aplicación. La funcionalidad restante tendrá que desarrollarse de un modo personalizado.

Refinamiento/contexto:
Subcondición 1: Ciertos componentes reutilizables fueron desarrollados por terceras personas sin el conocimiento de los estandares internos de diseño.
Subcondición 2: El estándar de diseño para interfaces de componentes **no** ha sido asentado y puede **no** ajustarse a ciertos componentes reutilizables existentes.
Subcondición 3: Ciertos componentes reutilizables han sido implementados en un lenguaje no soportado por el entorno para el que el sistema ha sido construido.

Reducción/Supervisión:
1. Contactar con terceras personas para determinar la conformidad con los estándares de diseño.
2. Presionar para completar los estándares de la interfaz; considerar la estructura de componentes cuando se decide el protocolo de la interfaz.
3. Comprobación para determinar los componentes en la categoria de subcondición 3; comprobación para determinar si se puede adquirir el soporte del lenguaje

Gestión/Plan de Contingencia/Acción:
Se calcula que la ER es de £20,200. Esta cantidad se coloca dentro del coste de contingencia del proyecto. La planificación del proyecto revisado asume que se tendrán que construir 18 componentes adicionales; por consiguiente se asignará el personal de acuerdo con las necesidades.
Acción: Las fases de reducción se llevarán a cabo a partir de 7/1/02

Estado actual:
5/12/02: Fases de reducción iniciadas.

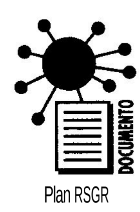
Autor: John Gagne	**Asignado:** B. Laster

FIGURA 6.5. Hoja de información de riesgo [WIL97].

La *seguridad del software y el análisis del peligro* son actividades para garantizar la calidad del software (Capítulo **8)** que se centra en la identificación y evaluación de peligros potenciales que pueden impactar al software negativamente y provocar que falle el sistema entero. Si se pueden identificar los peligros al principio del proceso de ingeniería del software, se pueden especificar características de diseño software que eliminen o controlen esos peligros potenciales.

6.8 EL PLAN RSGR

Se puede incluir una estrategia de gestión de riesgo en el plan del proyecto de software o se podrían organizar los pasos de gestión del riesgo en un Plan diferente de reducción, supervisión y gestión del riesgo (Plan RSGR). Todos los documentos del plan RSGR se llevan a cabo como parte del análisis de riesgo y son empleados por el jefe del proyecto como parte del Plan del Proyecto general.



Plan RSGR

Algunos equipos de software no desarrollan un documento RSGR formal. **Más** bien, cada riesgo se documenta utilizando una *hoja de información de riesgo (HIR)* [WIL97]. En la mayoría de los casos, la *HZR* se mantiene utilizando un sistema de base de datos, por lo que la creación y entrada de información, ordenación por prioridad, búsquedas y otros análisis pueden ser realizados con facilidad.. El formato de la HIR se muestra en la Figura 6.5.

Una vez que se ha desarrollado el plan RSGR y el proyecto ha comenzado, empiezan los procedimientos de reducción y supervisión del riesgo. Como ya hemos dicho antes, la reducción del riesgo es una actividad para evitar problemas. La supervisión del riesgo es una actividad de seguimiento del proyecto con tres objetivos principales: (1) evaluar cuando un riesgo previsto ocurre de hecho; (2) asegurarse de que los procedimientos para reducir el riesgo definidos para el riesgo en cuestión se están aplicando apropiadamente; y **(3)** recoger información que pueda emplearse en el futuro para analizar riesgos. En muchos casos, los problemas que ocurren durante un proyecto pueden afectar a más de un riesgo. Otro trabajo de la supervisión de riesgos es intentar determinar el *origen* -qué riesgo(s) ocasionaron tal problema a lo largo de todo el proyecto–.

RESUMEN

Cuando se pone mucho en juego en un proyecto de software el sentido común nos aconseja realizar un análisis de riesgo. Y sin embargo, la mayoría de los jefes de proyecto lo hacen informal y superficialmente, si es que lo hacen. El tiempo invertido identificando, analizando y gestionando el riesgo merece la pena por muchas razones: menos trastornos durante el proyecto, una mayor habilidad de seguir y controlar el proyecto y la confianza que da planificar los problemas antes de que ocurran.

El análisis de riesgos puede absorber una cantidad significativa del esfuerzo de planificación del proyecto. Pero el esfuerzo merece la pena. Por citar a Sun Tzu, un general chino que vivió hace 2.500 años, « Si conoces al enemigo y te conoces a ti mismo, no tendrás que temer el resultado de cien batallas». Para el jefe de proyectos de software, el enemigo es el riesgo.

REFERENCIAS

[AFC88] *Software Risk Ahatement,* AFCS/AFLC Pamphlet 800-45, U.S. Air Force, **30** de Septiembre 1988.

[BOE89] Boehm, B. W., *Software Risk Management,* IEEE Computes Society Press, 1989.

[CHA89] Charette, R. N., *Software Engineering Risk Analysis and Management,* McGraw-Hill/Intertext, 1989.

[CHA92] Charette, R. N., «Building Bridges Over Intelligent Rivers », *American Programmer,* vol. 5, n.º 7, Septiembre 1992, pp. 2-9.

[DRU75] Drucker, P., *Management,* W. Heinemann, Ltd., 1975.

[GIL88] Gilb, T., *Principies of Software Engineering Management,* Addison-Wesley, 1988.

[GLU94] Gluch, D.P., «A Construct for Describing Software Development Risk», CMU/SEI-94-TR-14, Software Engineering Institute, 1994.

[HAL98] Hall, E. M., *Managing Risk: Methods for Software Systems Development,* Addison Wesley, 1998.

[HIG95] Higuera, R.P, «Team Risk Management», *CrossTalk*, US Dept. of Defense, Enero 1995, pp. 2-4.

[KAR96] Karolak, D.W., ***Software Engineering Risk Management,*** IEEE Computer Society Press, 1996.

[KEI98] Keil, M. et al., «A Framework for Identifying Software Project Risks», CACM, vol 41, n.º11, Noviembre 1998, pp. 76-83.

[LEV95] Leveson, N. G., ***Safeware: System Safety and Computes,*** Adisson-Wesley, 1995.

[ROW88] Rowe, W. D., ***An Anatomy of Risk,*** Robert E. Krieger Publishing Co., Malabar, FL, 1988.

[SEI93] ***Taxonomy-Based Risk Identification***, Software Engineering Institute, CMU/SEI-93-TR-6, 1993.

[TH092] Thomsett, R., «The Indiana Jones School of Risk Management», ***American Programmer***, vol. 5, n.° 7, Septiembre 1992, pp. 10-18.

[WIL97] Williams, R.C., J.A.Walker y A.J. Dorofee, «Putting Risk Management into Practice», ***IEEE Software,*** Mayo 1997, pp. 75-81.

PROBLEMAS Y PUNTOS A CONSIDERAR

6.1. Proporcione cinco ejemplos de otros campos que ilustren los problemas asociados con una estrategia reactiva frente al riesgo.

6.2. Describa la diferencia entre «riesgos conocidos» y «riesgos predecibles».

6.3. Añada tres cuestiones ***o*** temas a cada una de las listas de comprobación de elementos de riesgo que se presentan en el sitio web SEPA.

6.4. Se le ha pedido que construya un software que soporte un sistema de edición de vídeo de bajo coste. El sistema acepta cintas de vídeo como entrada de información, almacena el vídeo en disco, y después permite al usuario realizar un amplio abanico de opciones de edición al vídeo digitalizado. El resultado (salida) se envía a una cinta. Realice una pequeña investigación sobre sistemas de este tipo, y después haga una lista de riesgos tecnológicos a los que se enfrentaría al comenzar un proyecto de este tipo.

6.5. Usted es el jefe de proyectos de una gran compañía de software. Se le ha pedido que dirija a un equipo que está desarrollando un software de un procesador de textos de «nueva generación» (ver Sección 3.4.2 para obtener una breve descripción). Construya una tabla de riesgo para el proyecto.

6.6. Describa la diferencia entre componentes de riesgo y controladores de riesgo.

6.7. Desarrolle una estrategia de reducción del riesgo y sus actividades específicas para tres de los riesgos señalados en la Figura 6.2.

6.8. Desarrolle una estrategia de supervisión del riesgo y sus actividades específicas para tres de los riesgos señalados en la Figura 6.2. Asegúrese de identificar los factores que va a supervisar para determinar si el riesgo se hace más o menos probable.

6.9. Desarrolle una estrategia de gestión del riesgo y sus actividades específicas para tres de los riesgos señalados en la Figura 6.2.

6.10. Trate de refinar tres de los riesgos señalados en la figura 6.2 y realice las hojas de información del riesgo para cada uno.

6.11. Represente 3 de los riesgos señalados en la figura 6.2 utilizando el formato CTC.

6.12. Vuelva a calcular la exposición al riesgo tratada en la Sección 6.4.2 donde el coste/LDC es £16 y la probabilidad es del 60 por 100.

6.13. ¿Se le ocurre alguna situación en la que un riesgo de alta probabilidad y gran impacto no formará parte de su plan RSGR?

6.14. Respecto a la referencia de riesgo mostrada en la Figura 6.4, ¿será siempre la curva un arco simétrico como el que aparece, o habrá situaciones en las que la curva esté más distorsionada? Si es así, sugiera un escenario en el que esto pueda ocurrir.

6.15. Realice una investigación sobre aspectos de seguridad del software y escriba una pequeña redacción sobre el tema. Desarrolle un buscador web para obtener información actual.

6.16. Describa cinco áreas de aplicación de software en las que la seguridad del software y el análisis de riesgo sean vitales.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Las lecturas sobre la gestión del riesgo del software se han expandido significativamente en los Últimos años. Hall **[HAL98]** presenta uno de los tratados más completos del tema. Karolak [KAR96] ha escrito una guía que introduce un modelo de análisis del riesgo fácil de usar con listas de comprobación y cuestionarios que merece la pena. Grey ha realizado una instantánea de la evaluación del riesgo ***(Practica1 Risk Assessmentfor Project Management, Wiley,** 1995).* Su tratado abreviado proporciona una buena introducción al tema. Otros libros que merecen la pena son:

Chapman, C.B., y S. Ward, ***Project Risk Management: Processes, Techniques and Insights***, Wiley, 1997.

Schuyler, J. R., ***Decision Analysis in Projects,*** Project Management Institute Publications, 1997.

Wideman, R. M. (editor), ***Project & Program Risk Management: A Guide to Managing Project Risk and Opportunities***, Project Management Institute Publications, 1998.

Capers Jones ***(Assesment and Control of Software Risks,*** Prentice-Hall, 1994) presenta un detallado estudio sobre ries-

gos del software que incluye información reunida de cientos de proyectos de software. Jones define 60 factores de riesgo que pueden afectar al resultado de los proyectos de software. Boehm [BOE89] sugiere unos excelentes cuestionarios y formatos de listas de comprobación que pueden resultar de valor incalculable a la hora de identificar riesgos. Charrette [CHA89] presenta un detallado tratado de los mecanismos de análisis de riesgo, recurriendo a la teoría de probabilidades y técnicas estadísticas para analizar riesgos. En un volumen adjunto, Charrette ***(Application Strategies for Risk Analysis,*** McGrawHill, 1990) analiza el riesgo en el contexto tanto del sistema como de la ingeniería del software y define unas estrategias pragmáticas para la gestión del riesgo. Gilb [GIL88] presenta un conjunto de «principios» (que son a menudo divertidos y algunas veces profundos) que pueden servir como una Útil directriz para la gestión del riesgo.

Los tratados de Marzo 1995 de ***American Programmer***, Mayo 1997 de ***IEEE Software,*** y Junio 1998 ***Cutter IT Journal*** están dedicados a la gestión del riesgo.

El Instituto de Ingeniería del Software ha publicado muchos informes y guías detallados sobre el análisis y gestión del riesgo. El Air Force Systems Command Pamphlet AFSCP 800-45 [AFC88] describe técnicas de identificación y reducción de riesgos. Todos los números de ***ACM Software Engineering Notes*** publican una sección titulada «Riesgos para el público» (editor, P. G. Neumann). Si quiere las últimas y mejores historias de horror de software, éste es el lugar adonde ir.

En Internet están disponibles una gran variedad de fuentes de información relacionadas con temas de análisis y gestión del riesgo. Se puede encontrar una lista actualizada con referencias a sitios (páginas) web que son relevantes para el riesgo en **http://www.pressman5.com.**

7 PLANIFICACIÓN TEMPORAL Y SEGUIMIENTO DEL PROYECTO

A finales de los años 60, un joven y brillante ingeniero fue elegido para «escribir» un programa de computadora para una aplicación de fabricación automática. El motivo para su selección fue sencillo. Era la única persona de su grupo técnico que había asistido a un seminario de programación de computadoras. Sabía todo sobre el lenguaje ensamblador y Fortran, pero nada sobre ingeniería del software y mucho menos sobre la planificación temporal y seguimiento del proyecto.

Su jefe le dio los manuales adecuados y una descripción verbal de lo que tenía que hacer. Se le informó de que el proyecto debía terminarse en dos meses.

Leyó los manuales, consideró su enfoque y empezó a escribir código. Después de dos semanas de trabajo, el jefe le llamó a su oficina y le preguntó qué tal iban las cosas.

«Muy bien», dijo el joven ingeniero con entusiasmo juvenil, «es más fácil de lo que pensaba. He terminado cerca del 75 por 100».

El jefe sonrió. «Es realmente estupendo», dijo, alentando al joven ingeniero a que siguiera trabajando del mismo modo. Acordaron que se reunirían otra vez en una semana.

Una semana más tarde el jefe llamó al ingeniero a su oficina y le preguntó: «¿Por dónde vamos?»

«Todo va muy bien», dijo el joven, «pero me he encontrado con unos pequeños obstáculos. Los solucionaré y volveré al ritmo de trabajo pronto.»

«¿Cómo ves las fechas límite de entrega?», preguntó el jefe.

«No hay problema», dijo el ingeniero, «estoy cerca de terminar el 90 por 100».

Si ha estado trabajando en el mundo del software durante algunos años, sabrá el final de la historia. No es una sorpresa que el joven ingeniero[1] se estancara en el 90 por 100 durante el resto del proyecto y que terminara (con la ayuda de otros) con un mes de retraso.

Esta historia se ha repetido docenas de miles de veces con los desarrolladores de software durante las últimas tres décadas. La gran pregunta es ¿por qué?

VISTAZO RÁPIDO

¿Qué es? Usted ha seleccionado un modelo de proceso adecuado, ha identificado las tareas de ingeniería del software que hay que llevar a cabo, ha estimado la cantidad de trabajo y el número de personas necesario, conoce las fechas límite de entrega e incluso ha considerado los riesgos. Ahora es el momento de unir todos los puntos. Esto es, tiene que crear una red de tareas de ingeniería del software que le permitan conseguir el trabajo realizado a tiempo. Una vez creada la red, tiene que asignar la responsabilidad para cada tarea, asegúrese de hacerlo y de adaptar la red antes de que los riesgos se conviertan en realidad. En resumidas cuentas, esto es la planificación temporal y el seguimiento del proyecto de software.

¿Quién lo hace? A nivel de proyecto, los gestores de proyectos de software, utilizan la información solicitada a los ingenieros de software. A nivel individual, los mismos ingenieros de software.

¿Por qué es importante? Para construir un sistema complejo, muchas tareas de ingeniería del software se realizan en paralelo y el resultado del trabajo desarrollado durante una tarea puede tener un gran efecto en el trabajo a realizar en otra tarea. Estas interdependencias son muy difíciles de comprender sin una planificación. También es virtualmente imposible evaluar el progreso en un proyecto de software normal o grande sin una planificación detallada.

¿Cuáles son los pasos? Las tareas de ingeniería del software dictadas por el modelo de proceso del software son refinadas por la funcionalidad a construir. El esfuerzo y la duración se asigna a cada tarea y se crea una red de tareas (tambien llamada red de actividades) que permita al equipo de software conseguir la fecha límite de entrega establecida.

¿Cuál es el producto obtenido? Se obtiene la planificación del proyecto e información relacionada.

¿Cómo puedo asegurar que lo he hecho correctamente? Una planificación adecuada requiere que (1) todas las tareas aparezcan en la red; (2) el esfuerzo y el tiempo se asigne inteligentemente a cada tarea; (3) las relaciones entre tareas estén indicadas correctamente; (4) los recursos sean asignados al trabajo a realizar, y (5) los hitos se sitúen rigurosamente espaciados para que se pueda seguir el progreso.

[1] Si se pregunta si esta historia es autobiográfica, ¡lo es!

7.1 CONCEPTOS BÁSICOS

Aunque hay muchas razones por las que el software se entrega tarde, la mayoría pertenecen a una o más de las siguientes causas:

- Una fecha límite de entrega poco realista, establecida por alguien que no pertenece al grupo de ingeniería del software e impuesta a los gestores y profesionales del grupo.
- Cambio de los requisitos del cliente que no se reflejan en los cambios de la planificación temporal.
- Una subestimación honesta de la cantidad de esfuerzo y/o el número de recursos que serán necesarios para hacer el trabajo.
- Riesgos predecibles y no predecibles que no se consideraron cuando comenzó el proyecto.
- Dificultades técnicas que no pudieron ser previstas por adelantado.
- Dificultades humanas que no pudieron ser previstas por adelantado.
- Falta de comunicación entre la plantilla del proyecto que causa retrasos.
- Falta de reconocimiento por parte de la gestión del proyecto de su retraso y falta de medidas para corregir el problema.



Los planificaciones temporales irracionales o excesivas son probablemente la influencia mas destructiva en el software.
Capers Jones

Las fechas límite de entrega agresivas (léase «poco realistas»), son un hecho consumado en el mundo del software. Algunas veces estas fechas límite se piden por motivos legítimos desde el punto de vista de la persona que las establece, pero el sentido común también dice que la legitimidad debe ser percibida por las personas que hacen el trabajo.

7.1.1. Comentarios sobre los «retrasos»

Napoleón dijo una vez: «Un comandante en jefe que acepta llevar a cabo un plan que considera defectuoso está cometiendo un error; debe exponer sus razones, insistir en cambiar el plan y finalmente presentar su dimisión antes de ser el instrumento de la destrucción de su ejército.» Duras palabras que muchos gestores de proyecto de software deberían considerar.

Las actividades de estimación y análisis de riesgos estudiadas en los Capítulos 5 y 6, y las técnicas de planificación temporal descritas en este capítulo, se implementan a menudo bajo la restricción de una fecha límite definida. Si las mejores estimaciones indican que la fecha límite es poco realista, un gestor de proyecto competente debería «proteger a su equipo de una presión [de la planificación temporal] innecesaria...[y] devolver la presión a quienes la originaron» [PAG85].



Adoro las fechas de entrega. Me gusta el sonido peculiar que emiten cuando se aproximan.
Douglas Adams

Para ilustrarlo, imagínese que un grupo de desarrollo de software ha sido comisionado para construir un controlador de tiempo real para un instrumento médico de diagnóstico que quiere introducirse en el mercado en nueve meses. Después de una cuidadosa estimación y análisis de riesgos, el gestor del proyecto de software llega a la conclusión de que el software, tal y como se ha pedido, requerirá 14 meses para crearlo con la plantilla de la que se dispone. ¿Cómo debe proceder el jefe del proyecto?

Es poco realista ir a la oficina del cliente (en este caso el cliente probable es mercadotecnia/ventas) y exigirle que se cambie la fecha de entrega. Las presiones externas del mercado han dictado la fecha y el producto debe entregarse. También es absurdo rechazar el trabajo (desde el punto de vista de la carrera profesional). Así pues, ¿qué hacer?

? ¿Qué deberíamos hacer cuando la gestión exige que cumplamos una fecha límite de entrega que es imposible?

Se recomiendan los siguientes pasos en esta situación:

1. Realice una estimación detallada usando información de proyectos anteriores. Determine el esfuerzo estimado y la duración del proyecto.
2. Empleando un modelo de proceso incremental (Capítulo 2), establezca una estrategia de desarrollo que proporcione una funcionalidad crítica mínima para la fecha límite impuesta, pero deje otras funcionalidades para más tarde. Documente el plan.
3. Reúnase con el cliente y (empleando la estimación detallada) explique por qué la fecha límite impuesta no es realista. Asegúrese de apuntar que todas las estimaciones se basan en proyectos del pasado. Asegúrese también de indicar la mejora de porcentaje que se requerirá para conseguir la fecha límite tal y como está ahora[2]. El siguiente comentario sería apropiado:

[2] Si la mejora de porcentaje es del 10 al 25 por ciento, puede ser posible, de hecho, terminar el trabajo. Pero si se necesita que la mejora del porcentaje en el rendimiento del equipo sea mayor del 50 por ciento. Esta es una previsión no realista.

«Creo que podemos tener un problema con la fecha de entrega del software de controlador XYZ. Les he entregado a cada uno de ustedes una descomposición abreviada de los ritmos de producción de proyectos anteriores y una estimación que hemos hecho de varias maneras diferentes. Se fijarán en que he asumido un 20 por 100 de mejora con respecto a los ritmos de producción del pasado, pero todavía obtenemos una fecha de entrega que está más bien a 14 meses de calendario que a 9 meses de la presente fecha.»

Referencia cruzada

Los modelos de procesos incrementales se estudian en el Capítulo 2.

4. Oferte la estrategia de desarrollo incremental como alternativa.

   «Tenemos pocas opciones y me gustaría que tomaran una decisión basándose en ellas. Primero, podemos aumentar el presupuesto y conseguir recursos adicionales de manera que tengamos la oportunidad de tener el trabajo hecho en nueve meses, pero entiendan que esto aumenta el riesgo de poca calidad debido a lo apretado de las fechas[3].

   Segundo, podemos quitar un número determinado de funciones software y capacidades de las que piden. Esto hará la versión preliminar del producto algo menos funcional, pero podemos anunciar la funcionalidad completa para lanzarla a los 14 meses. Tercero, podemos contradecir a la realidad y desear poder completarlo en 9 meses. Nos encontraremos con algo que no se pueda entregar a un cliente de manera alguna. La tercera opción, espero que estén de acuerdo, es inaceptable. Nuestra experiencia y nuestras mejores estimaciones dicen que es poco realista y una receta para el desastre.»

Habrá gruñidos, pero si se presentan estimaciones sólidas basadas en una buena información histórica, es probable que se opte por alguna versión negociada de las opciones 1 ó 2. La fecha límite no realista se desvanecerá.

7.1.2. Principios básicos

A Fred Brooks, el conocido autor de *The Mythical Man-Month* [BRO95], se le preguntó una vez cómo se retrasan las planificaciones temporales de los proyectos. Su respuesta fue tan simple como profunda: «Diariamente.»

La realidad de un proyecto técnico (tanto si implica la construcción de una planta hidroeléctrica o desarrollar un sistema operativo) es que hay que realizar cientos de pequeñas tareas antes de poder alcanzar el objetivo final. Algunas de estas tareas quedan fuera del camino principal y pueden completarse sin preocuparse del impacto en la fecha de terminación del proyecto. Otras tareas se encuentran en el «camino crítico»[4]. Si estas tareas «críticas» se retrasan, la fecha de terminación del proyecto entero se pone en peligro.



Las tareas requeridas para lograr el objetivo de los gestores del proyecto no se deberían realizar manualmente. Hay muchas herramientas excelentes de planificación de proyectos. Utilícelas.

El objetivo del gestor del proyecto es definir todas las tareas del proyecto, construir una red que describa sus interdependencias, identificar las tareas que son críticas dentro de la red y después hacerles un seguimiento para asegurarse de que el retraso se reconoce «de inmediato». Para conseguirlo, el gestor debe tener una planificación temporal que se haya definido con un grado de resolución que le permita supervisar el progreso y controlar el proyecto.

La planificación temporal de un proyecto de software es una actividad que distribuye el esfuerzo estimado a lo largo de la duración prevista del proyecto, asignando el esfuerzo a las tareas específicas de la ingeniería del software. Es importante resaltar, sin embargo, que la planificación temporal evoluciona con el tiempo. Durante las primeras etapas de la planificación del proyecto, se desarrolla una *planificación temporal macroscópica*. Este tipo de planificación temporal identifica las principales actividades de la ingeniería del software y las funciones del producto a las que se aplican. A medida que el proyecto va progresando, cada entrada en la planificación temporal macroscópica se refina en una *planificación temporal detallada*. Aquí, se identifican y programan las tareas del software específicas (requeridas para realizar una actividad).



Una planificación temporal demasiado optimista no produce planes reales más cortos, sino más largos.
Steve McConnell

La planificación temporal para proyectos de desarrollo de software puede verse desde dos perspectivas bastante diferentes. En la primera se ha establecido ya (irrevocablemente) una fecha final de entrega de un sistema basado en computadora. La organización del software está limitada a distribuir el esfuerzo dentro del marco de tiempo previsto. El segundo punto de vista de la planificación temporal asume que se han estudiado unos límites cronológicos aproximados pero que la fecha final será establecida por la organización de la ingenie-

[3] Puede añadir que agregar más personal no reducirá la planificación temporal proporcionalmente.

[4] El camino crítico se estudiará con más detalle, más adelante en este capítulo.

ría del software. El esfuerzo se distribuye para conseguir el mejor empleo de los recursos, y se define una fecha final después de un cuidadoso análisis del software. Desgraciadamente, la primera situación es más frecuente que la segunda.

Como todas las áreas de la ingeniería del software, la planificación temporal de proyectos de software se guía por unos principios básicos:

Compartimentación. El proyecto debe dividirse en un número de actividades y tareas manejables. Para llevar a cabo está compartimentación, se descomponen tanto el producto como el proceso (Capítulo 3).

Interdependencia. Se deben determinar las interdependencias de cada actividad o tarea compartimentada. Algunas tareas deben ocurrir en una secuencia determinada; otras pueden darse en paralelo. Algunas actividades no pueden comenzar hasta que el resultado de otras no esté disponible. Otras actividades pueden ocurrir independientemente.



Cuando realice una planificación temporal, compartimentalice el trabajo, represente interdependencias de tareas, asigne el esfuerzo y tiempo a cada tarea, defina responsabilidades para el trabajo a desarrollar, y defina los resultados y los hitos.

Asignación de tiempo. A cada tarea que se vaya a programar se le debe asignar cierto número de unidades de trabajo (por ejemplo, personas-día de esfuerzo). Además, a cada tarea se le debe asignar una fecha de inicio y otra de finalización que son función de las interdependencias y de si el trabajo se hará a tiempo total o tiempo parcial.

Validación de esfuerzo. Todos los proyectos tienen un número definido de miembros de la plantilla. A medida que se hace la asignación de tiempo, el gestor del proyecto debe asegurarse de que no se ha asignado un número de personas mayor que el de la plantilla en ese momento. Por ejemplo, considere un proyecto que tiene una plantilla asignada de tres miembros (por ejemplo, 3 personas-día están disponibles por día de esfuerzo asignado[5]). Un día cualquiera, se deben realizar siete tareas concurrentemente. Cada tarea requiere 0,50 personas-día de esfuerzo. Se ha asignado más esfuerzo del que pueden realizar las personas disponibles.

Responsabilidades definidas. Cada tarea que se programe debe asignarse a un miembro del equipo específico.

Resultados definidos. Cada tarea programada debería tener un resultado definido. Para los proyectos de software, el resultado es normalmente un producto (por ejemplo, el diseño de un módulo) o una parte de un producto. Los productos se combinan frecuentemente en entregas.

Hitos definidos. Todas las tareas o grupos de tareas deberían asociarse con un hito del proyecto. Se consigue un hito cuando se ha revisado la calidad de uno o más productos (Capítulo 8) y se han aceptado.

Cada uno de los principios anteriores se aplica a medida que evoluciona la planificación temporal del proyecto.

7.2 LA RELACIÓN ENTRE LAS PERSONAS Y EL ESFUERZO

En un proyecto de desarrollo de software pequeño una sola persona puede analizar los requisitos, realizar el diseño, generar código y realizar las pruebas. A medida que crece el tamaño del proyecto más personas se ven envueltas. (Raramente podemos permitirnos el lujo de enfocar el esfuerzo de diez personas-año con una persona trabajando durante diez años.)

Hay un mito común (estudiado en el Capítulo 1) que todavía creen muchos gestores responsables de esfuerzos de desarrollo del software: «Si se retrasa el programa, siempre podremos añadir más programadores y recuperar el ritmo más adelante en el proyecto.» Desgraciadamente, añadir gente tarde a un proyecto tiene a menudo un efecto negativo, provocando aún más retraso. El personal agregado debe aprenderse el sistema y la gente que les enseña es la misma que estaba trabajando. Mientras están enseñando no se trabaja, y el proyecto se retrasa todavía más.



Si debe añadir recursos a un proyecto con retraso, asegúrese de que les ha asignado trabajo altamente compartimentalizado.

Además del tiempo que se tarda en aprender el sistema, el involucrar más personas aumenta el número de vías de comunicación y la complejidad de la comunicación a lo largo del proyecto. Aunque la comunicación es absolutamente esencial para el éxito del desarrollo

[5] En realidad, se dispone de menos de tres personas-día debido a reuniones no mencionadas, enfermedades, vacaciones y otras razones. Para nuestro propósito, sin embargo, asumimos un 100 por 100 de disponibilidad.

del software, cada nueva vía de comunicación requiere un esfuerzo adicional y, por tanto, un tiempo adicional.

7.2.1. Un ejemplo

Considere cuatro ingenieros del software, cada uno capaz de producir 5000 LDC/año cuando trabajan en proyectos individuales. Cuando se pone a estos cuatro ingenieros en un proyecto de equipo, son posibles seis vías de comunicación potenciales. Cada vía de comunicación requiere un tiempo que podría de otra manera emplearse en desarrollar software. Asumiremos que la productividad del equipo (medida en LDC) se verá reducida en 250 LDC/año por cada vía de comunicación, debido al gasto indirecto asociado con la comunicación. Por tanto, la productividad del equipo es 20.000 —(250 × 6) = 18.500 LDC/año— un 7,5 por ciento menos de lo que podríamos esperar[6].

El proyecto de un año en el que está trabajando el equipo anterior se retrasa y a dos meses de la fecha de entrega se agregan dos personas adicionales al equipo. El número de vías de comunicación se dispara a 14. La productividad de la nueva plantilla es la equivalente a 840 × 2 = 1.680 LDC para los dos meses restantes antes de la entrega. La productividad del equipo es ahora 20.000 + 1.680 - (250 x 14) = 18.180 LDC/año.



La relación entre el número de personas que trabajan en un proyecto de software y la productividad total no es lineal.

Aunque el ejemplo anterior es una burda simplificación exagerada de las circunstancias del mundo real, sirve para ilustrar otro punto clave: la relación entre el número de personas que trabajan en un proyecto de software y la productividad global no es lineal.

7.2.2. Una relación empírica

Recordando la «ecuación del software» [PUT92] que se introdujo en el Capítulo 5, podemos demostrar la relación altamente no lineal entre el tiempo cronológico para completar un proyecto y el esfuerzo humano aplicado al proyecto. El número de líneas de código entregadas (sentencias fuente), L, está relacionado con el esfuerzo y el tiempo de desarrollo por la ecuación:

$$L = P \times E^{1/3}\, t^{4/3}$$

donde E es el esfuerzo de desarrollo en personas-mes; P es un parámetro de productividad que refleja una variedad de factores que llevan a un trabajo de ingeniería del software de alta calidad (los valores típicos de P se encuentran entre 2.000 y 12.000); y t es la duración del proyecto en meses.



Cuando la fecha límite de entrega es cada vez más ajustada, se alcanza un punto en el que el trabajo no se puede completar según la planificación, sin tener en cuenta el número de personas que lo estén realizando. Afronte la realidad y defina una nueva fecha de entrega.

Despejando la ecuación del software anterior, podemos llegar a la expresión del esfuerzo de desarrollo E:

$$E = L^3 / (P^3\, t^4) \qquad (7.1)$$

donde E es el esfuerzo invertido (en personas-año) durante el ciclo entero de la vida del desarrollo del software y del mantenimiento, y t es el tiempo de desarrollo en años. La ecuación de esfuerzo de desarrollo puede relacionarse con el coste de desarrollo con la inclusión de un factor de coste laboral del trabajo ($/personas-año).

Esto lleva a unos interesantes resultados. Considere un proyecto complejo de software de tiempo real estimado en 33.000 LDC, un esfuerzo de 12 personas-año. Si se han asignado ocho personas al equipo del proyecto, el proyecto puede estar terminado en 1,3 años. Sin embargo, si extendemos la fecha de finalización a 1,75 años, la naturaleza altamente no lineal del modelo descrito en la Ecuación (7.1) lleva a:

$$E = L^3 / (P^3\, t^4) \sim 3{,}8 \text{ personas-año}$$

Esto implica que extendiendo la fecha de finalización seis meses, ¡podemos reducir el número de personas desde ocho hasta cuatro! La validez de estos resultados está abierta a debate, pero la implicación es clara: se pueden obtener beneficios usando menos gente durante un período de tiempo algo mayor para realizar el mismo trabajo.

7.2.3. Distribución del esfuerzo

Cada una de las técnicas de estimación del proyecto de software tratadas en el Capítulo 5 lleva a estimaciones de las unidades de trabajo (por ejemplo, personas-mes) requeridas para completar un desarrollo de software. Una distribución recomendada de esfuerzo en las fases de

[6] Es posible presentar un argumento en contra: la comunicación, si es efectiva, puede aumentar la calidad del trabajo que se está realizando, reduciendo, por tanto, la cantidad de repasos y aumentando la productividad individual de los miembros del equipo. ¡Todavía no hay veredicto final!

definición y desarrollo se conoce normalmente como la regla 40-20-40[7]. El cuarenta por ciento de todo el esfuerzo o más se asigna a las tareas de análisis y diseño. Un porcentaje similar se aplica a las pruebas. Se puede deducir correctamente que no se insiste mucho en la creación de código (un 20 por 100 del esfuerzo).

? ¿Cuánto esfuerzo debería emplearse en cada una de las principales tareas de ingeniería del software?

Esta distribución de esfuerzo debería usarse como una directriz solamente. Las características de cada proyecto dictan la distribución del esfuerzo. El esfuerzo gastado en la planificación de un proyecto raramente supera más del 2 ó el 3 por 100, a no ser que el plan comprometa a una organización a grandes gastos por el gran riesgo. El análisis de los requisitos pueden comprender de un 10 a un 25 por 100 del esfuerzo del proyecto. El esfuerzo invertido en análisis o creación de prototipos debería crecer proporcionalmente con el tamaño y la complejidad del proyecto. Entre un 20 y un 25 por 100 del esfuerzo se aplica normalmente al diseño del software. También debe considerarse el tiempo empleado en la revisión del diseño y la repetición subsiguiente.

Debido al esfuerzo aplicado al diseño de software, el código debería realizarse con relativa poca dificultad. Se puede alcanzar un rango del 15 al 20 por 100 del esfuerzo global. Las pruebas y las subsiguientes depuraciones de errores pueden dar cuenta de entre un 30 y un 40 por 100 restante del esfuerzo de desarrollo del software. La importancia del software dicta a menudo la cantidad de pruebas que se requieren. Si el software se ve desde el punto de vista humano (es decir, el fallo del software puede generar pérdidas de vidas humanas), se pueden considerar incluso porcentajes más altos.

7.3 DEFINICIÓN DE UN CONJUNTO DE TAREAS PARA EL PROYECTO DE SOFTWARE

En el Capítulo 2 se describieron diferentes modelos de proceso. Estos modelos ofrecen paradigmas diferentes para el desarrollo del software. Independientemente de que un equipo de software elija un paradigma secuencial lineal, uno iterativo, uno evolutivo, uno concurrente o alguna combinación, el modelo de proceso está lleno de conjuntos de tareas que permiten al equipo del software definir, desarrollar y finalmente mantener el software de computadora.

No hay un único conjunto de tareas que sea apropiado para todos los proyectos. El conjunto de tareas que sería apropiado para un sistema grande y complejo sería considerado exagerado para un producto de software pequeño, relativamente sencillo. Por tanto, un proceso de software eficaz debería definir una colección de conjuntos de tareas, cada una diseñada para satisfacer las necesidades de diferentes tipos de proyectos.



Un «conjunto de tareas» es una colección de entregas, hitos y tareas de ingeniería del software.

Un conjunto de tareas es una colección de tareas de la ingeniería del software, hitos y entregas que deben realizarse para completar un proyecto particular. El conjunto de tareas a elegir debe proporcionar suficiente disciplina para alcanzar una alta calidad para el software. Pero, al mismo tiempo, no se debe cargar al equipo del proyecto con trabajo innecesario.

Los conjuntos de tareas se diseñan para acomodar diferentes tipos de proyectos y diferentes grados de rigor. Aunque es difícil desarrollar una completa taxonomía sobre los tipos de proyectos de software, la mayoría de las organizaciones de software encuentran proyectos de los siguientes tipos:

I. *Proyectos de desarrollo del concepto* que se inician para explorar algún nuevo concepto de negocios o aplicación de alguna nueva tecnología.

II. *Proyectos de desarrollo de una nueva aplicación* que se aceptan como consecuencia del encargo de un cliente específico.

III. *Proyectos de mejoras de aplicaciones* que ocurren cuando un software existente sufre grandes modificaciones de su funcionamiento, rendimiento o interfaces que son observables por el usuario final.

IV. *Proyectos de mantenimiento de aplicaciones* que corrigen, adaptan o amplían un software existente en métodos que pueden no ser obvios de forma inmediata para el usuario final.

V. *Proyectos de reingeniería* que se lleva a cabo con la intención de reconstruir un sistema existente (heredado) en su totalidad o en parte.

[7] Hoy en día, se recomienda normalmente aplicar al análisis y a las tareas de diseño de grandes proyectos de desarrollo de software más del 40 por 100 de todo el esfuerzo del proyecto. De ahí que el nombre de «40-20-40» no se aplique en un sentido estricto.

Incluso dentro de un tipo de proyecto simple, hay muchos factores que influyen en el conjunto de tareas a elegir. Cuando se toman en combinación, estos factores proporcionan una indicación del grado de rigor con el que debería aplicarse el proceso del software.

7.3.1. Grado de rigor



El conjunto de tareas crecerá en tamaño y complejidad al mismo tiempo que crece el grado de rigor.

Incluso para un proyecto de un tipo en particular, *el grado de rigor* con el que se aplica el proceso del software puede variar significativamente. El grado de rigor es función de muchas características del proyecto. Por ejemplo, los pequeños proyectos no críticos del negocio pueden ser tratados con algo menos de rigor que grandes y complejas aplicaciones críticas desde el punto de vista del negocio. Debe advertirse, no obstante, que todos los proyectos deben ejecutarse de manera que terminen en puntuales entregas de alta calidad. Se pueden definir cuatro grados de rigor:

Casual. Se aplican todas las actividades estructurales del proceso (Capítulo 2), pero sólo se requiere un conjunto de tareas mínimo. En general, las tareas protectoras se minimizarán y se reducirán los requisitos de documentación. Son aplicables todavía todos los principios básicos de la ingeniería del software.

Estructurado. Se aplicará la estructura del proceso a este proyecto. Se aplicarán las actividades estructurales y las tareas relativas apropiadas para el tipo de proyecto, así como las actividades protectoras necesarias para garantizar una alta calidad. Se llevarán a cabo SQA (GCS), documentación y tareas de medición de manera fluida.

Estricto. Se aplicará el proceso completo para este proyecto con un grado de disciplina tal que garantice una alta calidad. Se aplicarán todas las actividades protectoras y se producirá una robusta documentación.

Reacción rápida. Se aplicará la estructura del proceso a este proyecto, pero debido a una situación de emergencia[8], sólo se aplicarán aquellas tareas esenciales para mantener una alta calidad. Se realizará un «back-filling» (es decir, desarrollar un conjunto completo de documentación, realizar revisiones adicionales) después de entregar la aplicación/producto al cliente.



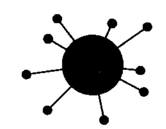
Si todo es una emergencia, habrá algo mal en su proceso del software o en las personas que lo dirigen o en ambos.

El gestor del proyecto debe desarrollar un enfoque sistemático para seleccionar el grado de rigor apropiado para cada proyecto. Para conseguirlo, se definen unos criterios de adaptación del proyecto y se calcula un valor selector del conjunto de tareas.

7.3.2. Definir los criterios de adaptación

Los criterios de adaptación se emplean para determinar el grado de rigor recomendado con el que el proceso del software debería aplicarse en un proyecto. Se definen once criterios de adaptación para proyectos de software [PRE99]:

- Tamaño del proyecto.
- Número potencial de usuarios.
- Importancia de la misión.
- Antigüedad de la aplicación.
- Estabilidad de los requisitos.
- Facilidad de comunicación cliente/desarrollador.
- Madurez de la tecnología aplicable.
- Limitaciones de rendimiento.
- Características empotradas/no empotradas.
- Personal del proyecto.
- Factores de reingeniería.



Modelo de proceso adaptable

A cada uno de los criterios de adaptación se le asigna un grado que va desde 1 hasta 5, donde 1 representa un proyecto en el que se requiere un pequeño subconjunto de tareas de proceso, y los requisitos generales metodológicos y de documentación son mínimos, y 5 representa un proyecto en el que se debería aplicar un conjunto completo de tareas de proceso y en el que los requisitos generales metodológicos y de documentación son sustanciales.

7.3.3. Cálculo del valor selector del conjunto de tareas

Para seleccionar el conjunto apropiado de tareas para un proyecto, se deberían seguir los siguientes pasos:

[8] Las situaciones de emergencia deberían ser raras (no debería ocurrir en más de un 10 por 100 de todo el trabajo realizado dentro del contexto de la ingeniería del software). Una emergencia no es lo mismo que un proyecto con apretadas limitaciones de tiempo.

1. Revise cada uno de los criterios de adaptación en la Sección 7.3.2 y asigne los grados apropiados (1 a 5) basándose en las características del proyecto. Estos grados deberían introducirse en la Tabla 7.1.

¿Cómo elegimos el conjunto de tareas apropiado para nuestro proyecto?

Criterios de Adaptación	Grado	Peso	Multiplicador de punto de entrada					Producto
			Concur.	Ndev.	Mejora	Mant.	Reing.	
Tamaño del proyecto	____	1,20	0	1	1	1	1	____
Número de usuarios	____	1,10	0	1	1	1	1	____
Importancia para el negocio	____	1,10	0	1	1	1	1	____
Antigüedad	____	0,90	0	1	1	0	0	____
Estabilidad de los requisitos	____	1,20	0	1	1	1	1	____
Facilidad de Comunicación	____	0,90	1	1	1	1	1	____
Madurez de la tecnología	____	0,90	1	1	0	0	1	____
Limitaciones de rendimiento	____	0,80	0	1	1	0	1	____
Empotrado/no empotrado	____	1,20	1	1	1	0	1	____
Personal del proyecto	____	1,00	1	1	1	1	1	____
Interoperabilidad	____	1,10	0	1	1	1	1	____
Factores de reingeniería	____	1,20	0	0	0	0	1	____
Selector de conjunto de tareas (SCT)								____

TABLA 7.1. CÁLCULO DEL SELECTOR DEL CONJUNTO DE TAREAS. UN EJEMPLO

2. Revise los factores de ponderación asignados a cada criterio. El valor de un factor de ponderación va desde 0.8 a 1.2 y proporciona una indicación de la relativa importancia de un criterio de adaptación en particular a los tipos de software desarrollados dentro del entorno local. Si son necesarias modificaciones para reflejar mejor las circunstancias locales, se deberían hacer.
3. Multiplique el grado introducido en la Tabla 7.1 por el **factor de ponderación** (peso) y por el **multiplicador de punto de entrada** del tipo de proyecto a realizar. El multiplicador de punto de entrada toma un valor entre 0 y 1, e indica la importancia del criterio de adaptación para el tipo de proyecto. El resultado del producto:

   Grado x **factor de ponderación** x **multiplicador de punto de entrada**

   se coloca en la columna **Producto** de la Tabla 7.1 para cada criterio de adaptación individual.
4. Calcule la media de todas las entradas en la columna **Producto** y ponga el resultado en el espacio marcado selector del conjunto de tareas (SCT). Este valor se usará para ayudarle a seleccionar el conjunto de tareas más apropiado para el proyecto.

Criterios de Adaptación	Grado	Peso	Multiplicador de punto de entrada					Producto
			Concur.	Ndev.	Mejora	Mant.	Reing.	
Tamaño del proyecto	2	1,2	____	1	____	____	____	2,4
Número de usuarios	3	1,1	____	1	____	____	____	3,3
Importancia para el negocio	4	1,1	____	1	____	____	____	4,4
Antigüedad	3	0,9	____	1	____	____	____	2,7
Estabilidad de los requisitos	2	1,2	____	1	____	____	____	2,4
Facilidad de Comunicación	2	0,9	____	1	____	____	____	1,8
Madurez de la tecnología	2	0,9	____	1	____	____	____	1,8
Limitaciones de rendimiento	3	0,8	____	1	____	____	____	2,4
Empotrado/no empotrado	3	1,2	____	1	____	____	____	3,6
Personal del proyecto	2	1,0	____	1	____	____	____	2,0
Interoperabilidad	4	1,1	____	1	____	____	____	4,4
Factores de reingeniería	0	1,2	____	0	____	____	____	0,0
Selector de conjunto de tareas (SCT)								**2,8**

TABLA 7.2. CÁLCULO DEL SELECTOR DEL CONJUNTO DE TAREAS. UN EJEMPLO

7.3.4. Interpretar el valor SCT y seleccionar el conjunto de tareas

Una vez que se ha calculado el selector del conjunto de tareas, se pueden utilizar las siguientes directrices para seleccionar el conjunto de tareas apropiado para un proyecto:

Valor del selector del conjunto de tareas	Grado de rigor
SCT < 1,2	Casual
1.0 < SCT < 3,0	Estructurado
SCT > 2,4	Estricto



Si el valor del selector del conjunto de tareas es un área solapada, normalmente es correcto elegir el menor grado de rigor formal, a no ser que el riesgo del proyecto sea alto.

El solapamiento en los valores SCT de un conjunto de tareas recomendado con otro está hecho a propósito y pretende ilustrar que no son posibles unas fronteras tan definidas cuando se seleccionan conjuntos de tareas. En el análisis final, el valor del selector del conjunto de tareas, la experiencia acumulada y el sentido común deben tenerse en cuenta en la elección del conjunto de tareas para el proyecto.

La Tabla 7.2 ilustra cómo podría calcularse el SCT para un hipotético proyecto. El gestor del proyecto selecciona los grados mostrados en la columna **Grado.** El tipo de proyecto es el desarrollo de una nueva aplicación. Por tanto, los multiplicadores de punto de entrada se seleccionan de la columna **NDev**. La entrada en la columna **Producto** se calcula empleando

Grado x **Peso** x **Multiplicador de punto de entrada Ndev(NewDevelop.)**

El valor de SCT (calculado corno la media de todas las entradas de la columna Producto) es 2,8. Usando los criterios estudiados anteriormente, el gestor tiene la opción de usar tanto el conjunto de tareas estructurado como el estricto. La decisión se toma una vez que se han considerado todos los factores del proyecto.

7.4 SELECCIÓN DE LAS TAREAS DE INGENIERÍA DEL SOFTWARE

Para desarrollar una planificación temporal del proyecto, se debe distribuir un conjunto de tareas a lo largo de la duración del proyecto. Como apuntamos en la Sección 7.3 el conjunto de tareas variará dependiendo del tipo de proyecto y del grado de rigor. Cada uno de los tipos de proyectos descritos en la Sección 7.3 puede enfocarse usando un modelo de proceso lineal secuencial e iterativo (por ejemplo, el modelo incremental o de creación de prototipos) o evolutivo (por ejemplo, el modelo en espiral). En algunos casos, un tipo de proyecto fluye suavemente hacia el siguiente. Por ejemplo, los proyectos de desarrollo de concepto que tienen éxito evolucionan a menudo en nuevos proyectos de desarrollo de aplicación. Cuando termina un proyecto de desarrollo de una nueva aplicación, empieza a veces un proyecto de mejora de la aplicación. Esta progresión es natural y predecible y ocurrirá sea cual sea el modelo de proceso que adopte una organización. Por consiguiente, las principales tareas de ingeniería del software descritas en las secciones siguientes son aplicables a todos los flujos del modelo de proceso. Como ejemplo, consideremos las tareas principales de ingeniería para los proyectos de desarrollo de concepto.

Un modelo de proceso adaptable (MPA) completo incluye una variedad de conjuntos de tareas y está disponible para su uso.

Los proyectos de desarrollo de concepto se inician cuando se debe explorar el potencial de alguna nueva tecnología. No hay certeza de que la tecnología sea aplicable, pero el cliente (por ejemplo, marketing) cree que existe un beneficio potencial. Los proyectos de desarrollo de concepto se enfocan aplicando las siguientes tareas principales:

Ámbito del concepto— determina el ámbito general del proyecto.

Planificación preliminar del concepto— establece la capacidad de la organización para llevar a cabo el trabajo implicado por el ámbito del proyecto.

Valoración del riesgo tecnológico— evalúa el riesgo asociado con la tecnología a implementar como parte del proyecto.

Prueba de concepto— demuestra la viabilidad de una nueva tecnología en el contexto del software.

Implementación del concepto— implementa la representación del concepto de manera que pueda revisarlo un cliente y se emplea para propósitos de «marketing» cuando hay que vender un concepto a otros clientes o departamentos de gestión.

Reacción del cliente ante el concepto— solicita la opinión del cliente sobre un nuevo concepto de tecnología y va encaminada a aplicaciones específicas del cliente.

No deberían producirse sorpresas si echamos un vistazo rápido a estas tareas principales. De hecho el flujo de las tareas de ingeniería del software para proyectos de desarrollo de concepto (y también para los otros tipos de proyectos) es poco más que sentido común.

El equipo del software debe entender lo que hay que hacer (ámbito); el equipo (o el gestor) debe determinar si hay alguien disponible para hacerlo (planificación); debe considerar los riesgos asociados con el trabajo (valoración del riesgo); probar la tecnología de alguna manera (prueba de concepto); e implementarlo en forma de prototipo de manera que el cliente pueda evaluarlo (implementación del concepto y evaluación del cliente). Finalmente, si el concepto es viable, se debe fabricar una versión de producción (traducción).

Es importante destacar que las actividades estructurales de las tareas de desarrollo de concepto son iterativas por naturaleza. Es decir, un proyecto de desarrollo de concepto cualquiera podría enfocar las tareas anteriores en varios incrementos planificados, cada uno diseñado para producir una entrega que pueda ser evaluada por el cliente.

Si se elige un modelo de proceso lineal, cada uno de estos incrementos se define en una secuencia repetitiva como se ilustra en la Figura 7.1. Durante cada secuencia, se aplican actividades protectoras (descritas en el Capítulo 2); se supervisa la calidad, y al final de cada secuencia, se produce una entrega. Con cada iteración, cada entrega debería converger hacia el producto final definido para la fase de desarrollo de concepto. Si se elige un modelo evolutivo, la distribución de tareas de la I.1 a la I.6 aparecería como se muestra en la Figura 7.2. Se pueden definir y aplicar de la misma manera las tareas principales de ingeniería del software para otros tipos de proyectos.

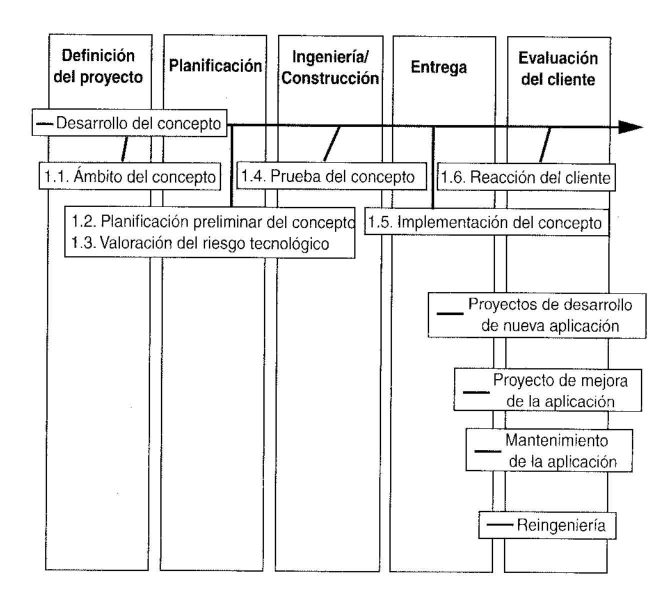


FIGURA 7.1. Tareas de desarrollo del concepto en un modelo secuencial lineal.

7.5 REFINAMIENTO DE LAS TAREAS PRINCIPALES

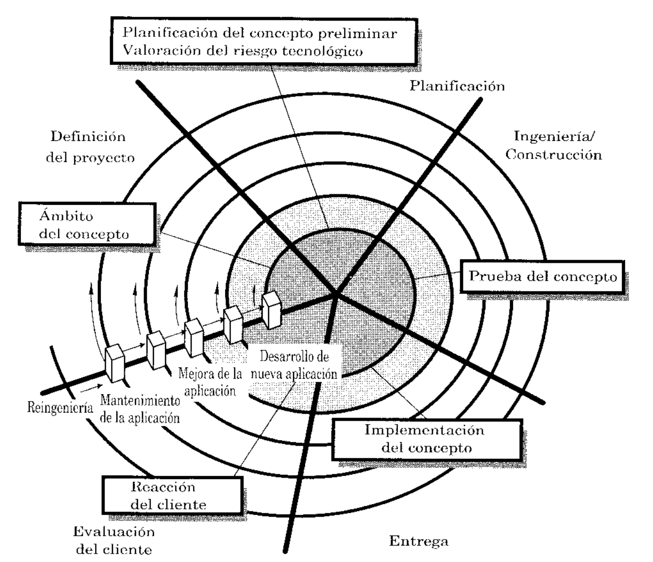


FIGURA 7.2. Tareas de desarrollo del concepto usando un modelo evolutivo.

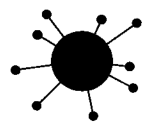
Las tareas principales descritas en la Sección 7.4 pueden emplearse para definir una planificación temporal macroscópica para el proyecto. Sin embargo, la planificación temporal macroscópica debe refinarse para crear una planificación temporal detallada del proyecto. El refinamiento se empieza tomando cada tarea principal y descomponiéndola en un conjunto de subtareas (con productos de trabajo e hitos relacionados).

Como ejemplo de descomposición de tarea, considere el **Ámbito del Concepto** como un concepto del proyecto de desarrollo, estudiado en la Sección 7.4.1. Se puede refinar la tarea empleando un formato de bosquejo, pero en este libro se emplea un enfoque de lenguaje de diseño de proceso para ilustrar el flujo de la actividad de ámbito del concepto:

```
Definición de tarea: Tarea I.1 Ámbito del concepto
I.1.1. Identificar la necesidad, los beneficios y
       clientes potenciales;
I.1.2. Definir el resultado/control deseado y las
       entradas que controlan la aplicación;
  Empezar la tarea I.1.2
    I.1.2.1. RTF: Revisar la descripción escrita de
             la necesidad [9];
    I.1.2.2. Obtener una lista de las entradas/sali-
             das visibles del cliente;
```

[9] RTF indica que se debe realizar una revisión técnica formal (Capítulo 8).

Caso de: procedimientos
Procedimientos = despliegue de la función calidad
reunirse con el cliente para aislar los principales requisitos del concepto;
entrevistar a los usuarios finales;
observar el enfoque actual al problema, proceso actual;
revisar requisitos y quejas anteriores;
Procedimientos = análisis estructurado
hacer una lista de los objetos de datos principales;
definir las relaciones entre los objetos;
definir los atributos de los objetos;
Procedimientos = visión del objeto
hacer una lista de las clases de problemas;
desarrollar la jerarquía de clases y las conexiones de clases;
definir los atributos de las clases;
fin caso
I.1.2.3. RTF: revisar las entradas/salidas con el cliente y adaptar según se requiera;
Fin de tarea Tarea I.1.2
I.1.3. Definir la funcionalidad/comportamiento para cada función principal que es desarrollada;
Empezar Tarea I.1.3
I.1.3.1. RTF: revisar los objetos de datos de entrada y salida obtenidos en la tarea
I.1.3.2. Obtener un modelo de funciones/comportamientos;
Caso de: procedimientos
Procedimientos = Despliegue de la función calidad
reunirse con el cliente para revisar los requisitos del concepto principal;
entrevistar a los usuarios finales;
observar el enfoque actual del problema, proceso actual;
desarrollar un esquema jerárquico de funciones/comportamientos;
Procedimientos = Análisis estructurado
obtener un diagrama de flujo de datos a nivel de contexto;
refinar el diagrama de flujo de datos para proporcionar más detalles;
escribir narrativas de proceso para las funciones a más bajo nivel de refinamiento;
Procedimientos = visión de objeto
definir operaciones/métodos relevantes para cada clase;
fin caso
I.1.3.3. Revisar funciones/comportamientos con el cliente y revisar según sea necesario;
Fin de tarea Tarea I.1.3
I.1.4. Aislar aquellos elementos de la tecnología a implementar en software;
I.1.5. Investigar la disponibilidad de información sobre el software existente;
I.1.6. Definir viabilidad técnica;
I.1.7. Hacer una estimación rápida del tamaño;
I.1.8. Crear una definición de ámbito;
Fin de tarea: Tarea I.1



El modelo de proceso adaptable (MPA) contiene una descripción del lenguaje de diseño de procesos completa para todas las tareas de ingeniería del software.

Las tareas y subtareas apuntadas en el refinamiento del lenguaje de diseño del proceso forman la base para una planificación temporal detallada de las actividades del ámbito del concepto.

7.6 DEFINIR UNA RED DE TAREAS

Las tareas y subtareas individuales tienen interdependencias basadas en su secuencia. Además, cuando hay más de una persona implicada en un proyecto de ingeniería del software, es probable que las actividades de desarrollo y tareas se realicen en paralelo. Cuando ocurre esto, las tareas concurrentes deben coordinarse de manera que estén finalizadas cuando tareas posteriores requieran su(s) resultado(s).



La red de tareas es un mecanismo útil para representar la dependencia entre tareas y para determinar el camino crítico.

Una *red de tareas*, también llamada *red de actividades*, es una representación gráfica del flujo de tareas de un proyecto. Se emplea a veces como el mecanismo a través del cual se introduce la secuencia de tareas y las dependencias en una herramienta de programación automática de la planificación temporal de un proyecto. En su forma más sencilla (la que se emplea en una planificación temporal macroscópica), la red de tareas muestra las tareas principales de ingeniería del software. La Figura 7.3 muestra una red de tareas esquemática para un proyecto de desarrollo de concepto.

La naturaleza concurrente de las actividades de ingeniería del software lleva a varios requisitos importantes de la planificación temporal. Como las tareas

paralelas ocurren asíncronamente, el planificador debe determinar las dependencias entre las tareas para garantizar un progreso continuo hasta su finalización. Además, el gestor del proyecto debería estar al tanto de las tareas que pertenecen al camino crítico. Es decir, tareas que deben finalizarse según la planificación temporal si se quiere que el proyecto en general se termine a tiempo. Estos aspectos se tratan con más detalle más adelante en este capítulo.

Es importante resaltar que la red de tareas mostrada en la Figura 7.3 es macroscópica. En una red de tareas detallada (el precursor de la planificación temporal detallada) cada actividad mostrada en la Figura 7.3 se expandiría. Por ejemplo, la Tarea I.1 se expandiría para mostrar todas las tareas detalladas en el refinamiento de tareas I.1 mostrado en la Sección 7.5.

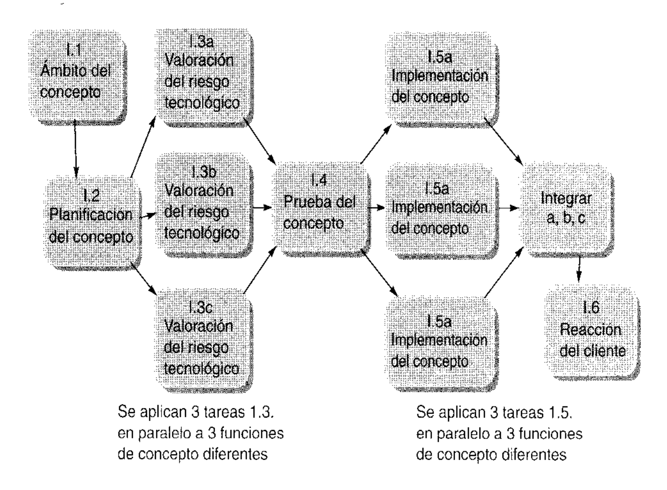


FIGURA 7.3. Una red de tareas para un proyecto de desarrollo de concepto.

7.7 LA PLANIFICACIÓN TEMPORAL

La planificación temporal de un proyecto de software no difiere mucho del de cualquier esfuerzo de ingeniería multitarea. Por tanto, se pueden aplicar herramientas de planificación temporal de proyectos y técnicas generales al software con una pequeña modificación en los proyectos del software.



Para los proyectos más sencillos, la planificación temporal debería realizarse con la ayuda de una herramienta de planificación temporal de proyectos.

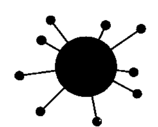
La *técnica de evaluación y revisión de programa (PERT)* y *el método del camino crítico (CPM)* [MOD83] son dos métodos de la planificación temporal de un proyecto que pueden aplicarse al desarrollo de software. Ambas técnicas son dirigidas por la información ya desarrollada en actividades anteriores de la planificación del proyecto:

- Estimaciones de esfuerzo.
- Una descomposición de la función del producto.
- La selección del modelo de proceso adecuado y del conjunto de tareas.
- La descomposición de tareas.

Las interdependencias entre las tareas deben definirse empleando una red de tareas. Las tareas, a veces denominadas *estructura de descomposición del trabajo* del proyecto (en inglés, WBS), se definen para el producto como un todo o para las funciones individuales.

Tanto PERT como CPM proporcionan herramientas cuantitativas que permiten al planificador del software : (1) determinar el *camino crítico*, la cadena de tareas que determina la duración del proyecto; (2) establecer las dimensiones de tiempo «más probables» para las tareas individuales aplicando modelos estadísticos, y (3) calcular las *limitaciones de tiempo* que definen una «ventana» de tiempo de una tarea determinada.

Los cálculos de las limitaciones de tiempo pueden ser muy útiles en la planificación temporal de proyectos de software. El retraso en el diseño de una función, por ejemplo, puede retardar el posterior diseño de otras funciones. Riggs [RIG81] describe importantes limitaciones de tiempo que pueden discernirse de una red PERT o CPM: (1) lo antes posible que puede empezar una tarea cuando las tareas precedentes se completen también lo antes posible; (2) lo más tarde que se puede empezar una tarea antes de que se retrase el tiempo mínimo para finalizar el proyecto; (3) la fecha más temprana de finalización —la suma de la fecha más temprana de inicio y la duración de la tarea—; (4) la fecha límite de finalización —la fecha más tardía de inicio sumada a la duración de la tarea—, y (5) el *margen total* —la cantidad de tiempo extra o atrasos permitidos en la planificación temporal de las tareas de manera que el camino crítico de la red se mantenga conforme a la planificación temporal—. Los cálculos de los tiempos límite llevan a la determinación del camino crítico y proporcionan al gestor un método cuantitativo para evaluar el progreso a medida que se completan las tareas.



Herramientas CASE para la planificación temporal y programación del proyecto.

Tanto PERT como CPM se han implementado en varias herramientas automáticas disponibles virtualmente para todos los ordenadores personales [THE93]. Tales herramientas son fáciles de usar y hacen asequibles los métodos de la planificación temporal descritos anteriormente a todos los gestores de proyectos de software.

7.7.1. Gráficos de tiempo

Cuando se crea una planificación temporal de un proyecto de software, el planificador empieza un conjunto de tareas (la estructura de descomposición del trabajo). Si se emplean herramientas automáticas, la descomposición del trabajo es introducida como una red de tareas o esquema de tareas. El esfuerzo, duración y fecha de inicio son las entradas de cada tarea. Además, se asignan las tareas a individuos específicos.



Un gráfico de tiempo le permite conocer qué tareas serán conducidas en un punto determinado en el tiempo.

Como consecuencia de esta entrada, se genera un *gráfico de tiempo*, también denominado *Gráfico Gantt*. Se puede desarrollar un gráfico de tiempo para todo el proyecto. Alternativamente, se pueden desarrollar diferentes gráficos para cada función del proyecto o para cada individuo que trabaje en el proyecto.

Tareas	Semana 1	Semana 2	Semana 3	Semana 4	Semana 5
I.1.1 Identificar necesidades y beneficios					
Reunirse con los clientes					
Identificar las necesidades y las limitaciones del proyecto					
Establecer la declaración de producto					
Hito: declaración de producto definida					
I.1.2 Definir las salidas/control/entradas deseadas (SCE)					
Alcance de las funciones del teclado					
Alcance de las funciones de entrada de voz					
Alcance de los modos de interacción					
Alcance de documento de diagnósticos					
Alcance otras funciones WP					
Documento SCE					
RTF: revisar el SCE con el cliente					
Revisar el SCE según se requiera					
Hito: SCE definido					
I.1.3 Definir la funcionalidad/comportamiento					
Definir las funciones del teclado					
Definir las funciones de entrada de voz					
Describir los modos de interacción					
Describir las comprobaciones de ortografía/gramática					
Describir otras funciones WP					
RTF: Revisar la definición SCE con el cliente					
Revisar según sea necesario					
Hito: Definición de SCE completa					
I.1.4 Aislar los elementos software					
Hito: Elementos software definidos					
I.1.5 Investigar la disponibilidad del software existente					
Investigar los componentes de edición de texto					
Investigar los componentes de la entrada de voz					
Investigar los componentes de la administración de archivos					
Investigar los componentes de la comprobación de ortografía y gramática					
Hito: Componentes reutilizables identificados					
I.1.6 Definir la viabilidad técnica					
Evaluar la entrada de voz					
Evaluar la comprobación de gramática					
Hito: Viabilidad técnica valorada					
I.1.7 Hacer una estimación rápida del tamaño					
I.1.8 Crear una definición de ámbito					
Revisar el documento de ámbito con el cliente					
Revisar el documento según se requiera					
Hito: Documento de ámbito completo					

FIGURA 7.4. Un ejemplo de gráfico de tiempo.

La Figura 7.4 ilustra el formato de un gráfico de tiempo. Muestra una parte de la planificación temporal de un proyecto de software que enfatiza la tarea de ámbito del concepto (Sección 7.5) para un nuevo producto de software de procesador de textos. Todas las tareas del proyecto (para ámbito del concepto) se listan en la columna de la izquierda. Las barras horizontales indican la duración de cada tarea. Cuando aparecen múltiples barras al mismo tiempo en la planificación temporal, implican concurrencia de tareas. Los rombos indican hitos.

Una vez que se ha introducido la información necesaria para generar el gráfico de tiempo, la mayoría de las herramientas de la planificación temporal de proyectos de software producen *tablas de proyecto* —un listado tabular de todas las tareas del proyecto, sus fechas previstas y reales de inicio y finalización, e información varia relativa al tema— (Fig. 7.5). Empleando las tablas junto con los gráficos de tiempo, le permiten al gestor del proyecto hacer un seguimiento del progreso.

7.7.2. Seguimiento de la planificación temporal

La planificación temporal del proyecto le proporciona al gestor un mapa de carreteras. Si se ha desarrollado apropiadamente, define las tareas e hitos que deben seguirse y controlarse a medida que progresa el proyecto. El seguimiento se puede hacer de diferentes maneras:

- realizando reuniones periódicas del estado del proyecto en las que todos los miembros del equipo informan del progreso y de los problemas;
- evaluando los resultados de todas las revisiones realizadas a lo largo del proceso de ingeniería del software;



La regla básica de los informes de estado del software puede resumirse en una frase: «¡ninguna sorpresa!».
Capers Jones

- determinando si se han conseguido los hitos formales del proyecto (los rombos mostrados en la Fig. 7.4) en la fecha programada;
- comparando la fecha real de inicio con las previstas para cada tarea del proyecto listada en la tabla del proyecto (Fig. 7.5);
- reuniéndose informalmente con los profesionales del software para obtener sus valoraciones subjetivas del progreso hasta la fecha y los problemas que se avecinan, o
- utilizando el análisis del valor ganado (Sección 7.8) para evaluar el progreso cuantitativamente.



El mejor indicador del progreso es la finalización y la revisión exitosa de un producto de software.

El control lo usa el gestor para administrar los recursos del proyecto, enfrentarse a los problemas y dirigir al personal del proyecto. Si las cosas van bien (es decir, el proyecto va según la planificación temporal y dentro del presupuesto, las revisiones indican que se está haciendo un progreso real y que se están alcanzando los hitos), el control es liviano. Pero cuando aparecen los problemas, el gestor debe ejercer el control para solucionarlos tan pronto como sea posible. Una vez que el problema se ha diagnosticado[10], se pueden concentrar recursos adicionales en el área del problema: se puede redistribuir la plantilla, o se puede redefinir la planificación temporal del proyecto.

Cuando se enfrentan a la presión de una fecha de entrega muy ajustada, los gestores de proyecto utilizan a veces una planificación temporal de proyecto y una técnica de control denominada *time-boxing* (tiempo encajonado) [ZAH95]. Esta estrategia reconoce que quizás no se pueda entregar el producto completo para la fecha límite predefinida. Por tanto, se elige un paradigma incremental del software (Capítulo 2) y se crea una planificación temporal para cada entrega de un incremento.

Las tareas asociadas con cada incremento se encajonan en el tiempo. Esto significa que la planificación temporal para cada tarea se ajusta trabajando hacia atrás desde la fecha de entrega para cada incremento. Se pone una «caja» alrededor de cada tarea. Cuando una tarea alcanza el límite de su caja de tiempo (más o menos un 10 por 100), se termina el trabajo y se empieza la siguiente tarea.

La primera reacción frente al enfoque de encajonamiento de tiempo es a menudo negativa: «Si no se ha terminado el trabajo, ¿cómo podemos proseguir?» La respuesta se encuentra en la manera en que se realiza el trabajo. Cuando se llega al límite de la caja de tiempo, es probable que se haya completado el 90 por 100 de la tarea[11]. El restante 10 por 100, aunque importante, puede (1) retrasarse hasta el siguiente incremento, o (2) completarse más tarde si es necesario. En vez de estar «estancado» en una tarea, el proyecto progresa hacia la fecha de entrega.

[10] Es importante resaltar que un retraso del programa es un síntoma de algún problema oculto. El papel del gestor es diagnosticar cuál es el problema y actuar para corregirlo.

[11] Un cínico podría recordar el dicho: «El primer 90 por 100 de un sistema se lleva el 10 por 100 del tiempo. El último 10 por 100 del sistema se lleva el 90 por 100 del tiempo.»

Tareas de trabajo	Inicio previsto	Inicio real	Terminación prevista	Terminación real	Personas asignadas	Esfuerzo asignado	Observaciones
I.1.1 Identificar necesidades y beneficios							Es previsible que requiera más esfuerzo/ tiempo
Reunirse con los clientes	Sem 1,d1	Sem 1,d1	Sem 1,d2	Sem 1,d2	BLS	2 p-d	
Identificar las necesidades y limitaciones del proyecto	Sem 1,d2	Sem 1,d2	Sem 1,d2	Sem 1,d2	JPP	1 p-d	
Establecer la declaración del producto	Sem 1,d3	Sem 1,d3	Sem 1,d3	Sem 1,d3	BLS/JPP	1 p-d	
Hito: declaración de producto definida	Sem 1,d3	Sem 1,d3	Sem 1,d3	Sem 1,d3			
I.1.2 Definir las salidas/control/entradas deseadas (SCE)							
Ámbito de las funciones del teclado	Sem 1,d4	Sem 1,d4	Sem 2,d2		BLS	1,5 p-d	
Ámbito de las funciones de entrada de voz	Sem 1,d3	Sem 1,d3	Sem 2,d2		JPP	2 p-d	
Ámbito de los modos de interacción	Sem 2,d1		Sem 2,d3		MLL	1 p-d	
Ámbito de los diagnósticos del documento	Sem 2,d1		Sem 2,d2		BLS	1,5 p-d	
Ámbito de otras funciones WP	Sem 1,d4	Sem 1,d4	Sem 2,d3		JPP	2 p-d	
Documento SCE	Sem 2,d1		Sem 2,d3		MLL	3 p-d	
RTF : revisar SCE con el cliente	Sem 2,d3		Sem 2,d3		Todos	3 p-d	
Revisar SCE según se requiera;	Sem 2,d4		Sem 2,d4		Todos	3 p-d	
Hito: SCE definido	Sem 2,d5		Sem 2,d5				
I.1.3 Definir la funcionalidad/comportamiento							

FIGURA 7.5. Un ejemplo de tabla de proyecto.

7.8 ANÁLISIS DE VALOR GANADO

En la sección 7.7.2, ya discutimos un número de aproximaciones cualitativas al seguimiento del proyecto. Cada una de ellas proporciona al gestor del proyecto una indicación del progreso, pero teniendo en cuenta que esta evaluación proporcionada de la información es en cierto modo subjetiva. Ahora bien, ¿es razonable preguntarse si existe una técnica cuantitativa para evaluar el progreso a medida que el equipo de software avanza a través de las tareas de trabajo asignadas en la planificación del proyecto? En efecto, una técnica para desarrollar análisis cuantitativo del progreso realizado existe. Es denominada *análisis de valor ganado (AVG)*.



El valor ganado proporciona una indicación cuantitativa del progreso.

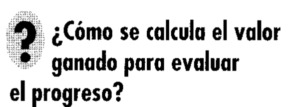
Humphrey [HUM95] estudia el valor ganado de la manera siguiente:

> El sistema de valor ganado proporciona una escala de valor común para cada tarea [proyecto de software], independientemente del tipo de trabajo que esté siendo llevado a cabo. Se estiman entonces el total de horas para realizar el proyecto completo y a cada tarea se le da un valor ganado basado en su porcentaje estimado respecto al total.

Dicho de forma más simple, el valor ganado es una medida del progreso. Nos permite evaluar el «porcentaje de realización» de un proyecto utilizando el análisis cuantitativo más que la opinión particular que de ello tengamos. En efecto, Fleming y Koppleman [FLE98] aducen que el análisis de valor ganado «proporcionan unas lecturas exactas y fiables del desarrollo desde estados iniciales como cuando tan sólo se haya realizado un 15 por ciento del proyecto».

Para determinar el valor ganado se desarrollan los siguientes pasos:

1. *El coste presupuestado del trabajo planificado (CPTP)* se determina para cada tarea de trabajo que se representa en el plan. Durante la actividad de estimación (Capítulo 5), el trabajo (en personas-hora, personas-día) de cada tarea de ingeniería del software es convenientemente planificada. Por consiguiente, CPTPi es el trabajo que se ha planificado para una cierta tarea i. Para determinar el progreso en un punto dado a lo largo de la planificación del proyecto, el valor de CPTP es la suma de los valores CPTPi para todas las tareas del trabajo que deberían haber sido completadas en ese momento en el plan del proyecto.



¿Cómo se calcula el valor ganado para evaluar el progreso?

2. Los valores CPTP para todas las tareas del trabajo se suman para obtener el *presupuesto a la terminación* que se denomina *PAT*. Por consiguiente,

$$PAT = \sum (CPTP_k) \text{ para todas las tareas } k$$

3. A continuación, se calcula el valor para *el coste presupuestado del trabajo desarrollado (CPTD)*. El valor para CPTD es la suma de los valores CPTP para todas las tareas de trabajo que hayan sido realmente terminadas en un punto determinado de la planificación del proyecto.

Wilkens [WIL99] afirma que «la distinción entre el CPTP y el CPTD es que el primero representa el presupuesto de las actividades que estaban planificadas para ser completadas, y el último representa el presupuesto de las actividades que realmente estaban acabadas».

Dados los valores para CPTP, PAT, CPTD, pueden ser calculados los siguientes indicadores de progreso:

Índice de desarrollo de planificación, IDP = CPTD/CPTP
Varianza de la planificación, VP = CPTD – CPTP

IDP es una indicación de la eficiencia con que el proyecto está utilizando los recursos de la planificación. Un valor IDP cercano a 1.0 indica una ejecución eficiente de la planificación del proyecto. VP es simplemente una indicación absoluta de la varianza de la planificación prevista.

Porcentaje planificado para terminar = CPTD/PAT

proporciona una indicación del porcentaje de trabajo que debería estar terminado en el instante *t*.

Porcentaje completado = CPTP/PAT

proporciona una indicación cuantitativa del «grado de avance en la realización en tanto por ciento» del proyecto en un instante determinado de tiempo *t*.

Es también posible calcular el *coste real de trabajo realizado, CRTR*. El valor para CRTR es la suma del esfuerzo realmente desarrollado en tareas de trabajo que hayan sido realizadas en un instante de tiempo de la planificación del proyecto. Es entonces posible calcular:

Índice de desarrollo del coste, IDC = CPTP/CRTR
Varianza del coste, VC = CPTP – CRTR



Se puede encontrar un gran conjunto de recursos para el análisis del valor ganado (bibliografía completa, documentos, enlaces) en **www.acq.osd.mil/pm/.**

Un valor de IDC cercano a 1.0 proporciona una indicación evidente de que el proyecto está dentro del presupuesto que para él se ha definido. VC es una indicación absoluta de los ahorros en coste (en relación con los costes planificados) o de las carencias en una etapa particular del proyecto.

Como ya ocurrió en la aparición del radar, el análisis del valor ganado aclara las dificultades de planificación antes de que ellas puedan aparecer. Esto permite al gestor del proyecto de software tomar las acciones correctivas adecuadas antes de que la crisis del proyecto estalle.

7.9 SEGUIMIENTO DEL ERROR

A través del proceso de software, un equipo que se dedica al proyecto crea productos de trabajo (por ejemplo, especificaciones de los requisitos o prototipos, documentos de diseño, código fuente). Pero este equipo también crea (y corrige afortunadamente) errores asociados con cada producto de trabajo realizado. Si las medidas relacionadas con los errores y las métricas resultantes son registradas a lo largo de muchos proyectos de software, un gestor de proyectos puede utilizar estos datos como una línea base para comparar esto con los datos de errores recogidos en tiempo real. El seguimiento del error puede utilizarse como una medida para evaluar el estado del proyecto actual.



El seguimiento del error nos permite comparar el trabajo actual con esfuerzos pasados y proporciona una indicación cuantitativa de la calidad del trabajo que estemos realizando.

En el Capítulo 4, el concepto de eficiencia de eliminación de defectos ya fue discutido. Para recordarlo brevemente, el equipo de software desarrolla revisiones técnicas formales (y, más tarde, comprobaciones) para encontrar y corregir los errores, *E*, en productos realizados durante las tareas de ingeniería de software. Cualquiera de los errores que no se hayan descubierto (pero se han encontrado en tareas posteriores) se considera que son defectos llamados *D*. La eficiencia de eliminación de defectos (Capítulo 4) ha sido definida como:

$$EED = E/(E+D)$$

EED es una métrica de proceso que proporciona una indicación clara de la efectividad de las actividades de garantía de calidad, pero EED y el cálculo de errores y defectos asociados con ella puede ser también usada para ayudar al gestor de proyectos a determinar el progreso que se está realizando a medida que el proyecto de software se mueve a través de las tareas de trabajo planificadas.

Asumamos que una organización de software ha registrado datos de errores y defectos durante los 24 meses pasados y ha desarrollado promedios para las métricas siguientes:

- Errores por página de especificación de requisitos, *Ereq*
- Errores por componentes a nivel de diseño, *Ediseño*
- Errores por componente a nivel de código, *Ecodigo*
- EED-análisis de requisitos
- EED-diseño arquitectónico
- EED-diseño a nivel de componentes
- EED-codificación

A medida que el proyecto progresa a través de cada paso de la ingeniería de software, el equipo de software registra e informa del número de errores encon-

trados en los requisitos, diseño y revisiones de código. El gestor del proyecto calcula los valores actuales para *Ereq, Ediseño* y *Ecodigo*. Estos son después comparados con los promedios de proyectos anteriores. Si los resultados actuales discrepan en más de un 20 por 100 de dicho promedio, ello puede ser causa de preocupación, y debe ser objeto de una investigación posterior.



Cuanto más cuantitativo sea su enfoque para el seguimiento y control del proyecto, estará probablemente más preparado para prevenir problemas potenciales y responder ante ellos de un modo proactivo. Utilice métricas de seguimiento y de valor ganado.

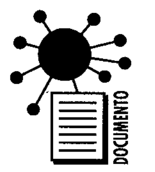
Por ejemplo, si *Ereq* =2.1 en el proyecto X, y el promedio de la organización es 3.6, es posible uno de estos dos escenarios: (1) que el equipo de software haya desarrollado el trabajo preferente de desarrollar las especificaciones de requisitos o (2) que el equipo haya prestado una atención secundaria en la aproximación a la revisión. Si el segundo escenario parece más probable, el gestor de proyecto debería adoptar los pasos necesarios para construir un tiempo[12] de diseño adicional dentro del plan con el fin de acomodar los defectos de requisitos que probablemente hayan podido propagarse a través de la actividad propia de diseño.

La métrica de seguimiento de errores descrita anteriormente puede ser utilizada también para los recursos de comprobación y/o revisión de objetivos. Por ejemplo, si un sistema se compone de 120 componentes, pero 32 de dichos componentes muestran valores *Ediseño* que tengan una varianza sustancial a partir del promedio, el gestor del proyecto puede elegir dedicar recursos de revisión de código a los 32 componentes y permitir a otros pasar a su comprobación con una revisión de código. Aunque todos los componentes deberían ser sometidos a una revisión de código en una supuesta revisión ideal, para los proyectos que se hayan pasado de presupuesto puede ser un medio efectivo realizar una aproximación selectiva (revisando sólo aquellos módulos que tengan una calidad dudosa basándose en el valor *Ediseño*) con el fin de recuperar el tiempo perdido y/o ahorrar los costes.

7.10 EL PLAN DEL PROYECTO

Cada paso en el proceso de ingeniería del software debería obtener una entrega que pueda revisarse y que pueda hacer de fundamento para los siguientes pasos. El *Plan del Proyecto de Software* se produce a la culminación de las tareas de planificación. Proporciona información básica de costes y planificación temporal que será empleada a lo largo del proceso de software.

El Plan del Proyecto de Software es un documento relativamente breve dirigido a una audiencia diversa. Debe (1) comunicar el ámbito y recursos a los gestores del software, personal técnico y al cliente; (2) definir los riesgos y sugerir técnicas de aversión al riesgo; (3) definir los costes y planificación temporal para la revisión de la gestión; (4) proporcionar un enfoque general del desarrollo del software para todo el personal relacionado con el proyecto, y (5) describir cómo se garantizará la calidad y se gestionan los cambios.



Plan del Proyecto de Software

Es importante señalar que el *Plan del Proyecto del Software no* es un documento estático. Esto es, el equipo del proyecto consulta el plan repetidamente —actualizando riesgos, estimaciones, planificaciones e información relacionada— a la vez que el proyecto avanza y es más conocido.

RESUMEN

La planificación temporal es la culminación de una actividad de planificación, componente primordial de la dirección de proyectos de software. Cuando se combinan métodos de estimación y análisis de riesgo, la planificación temporal se convierte en un mapa de carreteras a seguir por el gestor del proyecto.

La planificación temporal empieza con la descomposición del proceso. Las características del pro-

[12] En realidad, el tiempo extra será gastado en volver a trabajar en los defectos de requisitos, pero este trabajo tendrá lugar cuando se emprenda el trabajo de diseño.

yecto se emplean para adaptar un conjunto de tareas apropiado al trabajo a realizar. Una red de tareas muestra todas las tareas de ingeniería, sus dependencias con otras tareas y sus duraciones previstas. La red de tareas se usa para calcular el camino crítico del proyecto, un gráfico de tiempo e información diversa del proyecto. El gestor del proyecto puede seguir y controlar todos los pasos del proceso de software usando la planificación temporal como directriz.

REFERENCIAS

[BR095] Brooks, M., *The Mythical Man-Month*, ed. Aniversario, Adison-Wesley, 1995.

[FLE98] Fleming, Q.W., y J.M. Koppelman, «Earned Value Project Management», *Crosstalk*, vol.11, n.º 7, Julio 1998, p. 19.

[HUM95] Humphrey, W., *A Discipline for Software Engineering*, Addison-Wesley, 1995.

[MOD83] Moder, J. J., C. R. Philips y E. W. Davis, *Project Management with CPM, PERT and Precedence Diagramming*, 3.ª ed., VanNostrad Reinhold, 1983.

[PAG85] Page-Jones, M., *Practical Project Management*, Dorset House, 1985, pp. 90-91.

[PRE99] Pressman, R. S., *Adaptable Process Model*, R. S. Pressman&Associates, 1999.

[PUT92] Putman, L., y W. Myers, *Measures for Excellence*, Yourdon Press, 1992.

[RIG81] Riggs, J., *Production Systems Planning, Analysis and Control*, 3.ª ed., Wiley, 1981.

[THE93] Thé, L., «Project Management Software That's IS Friendly», *Datamation*, Octubre 1993, pp. 55-58.

[WIL99] Wilkens, T.T., «Earned Value, Clear and Simple», Primavera Systems, Inc, Abril 1999, p. 2.

[ZAH95] Zahniser, R., «Time-boxing for Top Team Performance», *Software Development*, Marzo 1995, pp. 34-38.

PROBLEMAS Y PUNTOS A CONSIDERAR

7.1. Las fechas límite «irracionales» son un hecho consumado del negocio del software. ¿Cómo procedería si se encuentra en esta situación?

7.2. ¿Cuál es la diferencia entre una planificación temporal macroscópica y una detallada? ¿Es posible dirigir un proyecto si sólo se desarrolla una planificación temporal macroscópica? ¿Por qué?

7.3. ¿Hay algún caso en el que un hito de un proyecto no está sujeto a una revisión? Si es así, proporcione uno o más ejemplos.

7.4. En la Sección 7.2.1, se presenta un ejemplo de «sobrecarga de comunicaciones» que puede ocurrir cuando mucha gente trabaja en un proyecto de software. Desarrolle un ejemplo que ilustre cómo unos ingenieros expertos, y con buenas experiencias en ingeniería del software y que utilizan revisiones técnicas formales, pueden mejorar el ritmo de producción de un equipo (comparado con la suma de los ritmos individuales de producción). Pista: puede asumir que las revisiones reducen el retrabajo y que estas repeticiones del trabajo suponen de un 20 a un 40 por 100 del tiempo de una persona.

7.5. Aunque agregar gente a un proyecto atrasado lo puede retrasar aún más, hay circunstancias en las que no es verdad. Descríbalas.

7.6. La relación entre el personal y el tiempo no es lineal. Usando la ecuación del software de Putman (descrita en la Sección 7.2.2), desarrolle una tabla que relacione el número de gente con la duración del proyecto para un proyecto de software que requiera 50.000 LDC y 15 personas-año de esfuerzo (el parámetro de productividad es 5.000 y B = 0,37). Asuma que el software debe entregarse en 24 meses, más/menos 12 meses.

7.7. Asuma que ha sido contratado por una universidad para desarrollar un *sistema interactivo para apuntarse a cursos (OLCRS)*. Primero, actúe como el cliente (¡si usted es un estudiante, le resultará fácil!) y especifique las características de un buen sistema. [Alternativamente, su profesor le proporcionará un conjunto de requisitos preliminares para el sistema.] Usando los métodos de estimación estudiados en el Capítulo 5, desarrolle una estimación de esfuerzo y duración para un OLCRS. Sugiera cómo:

a) definiría las actividades paralelas de trabajo durante el proyecto OLCRS;

b) distribuiría el esfuerzo a lo largo del proyecto;

c) establecería hitos para el proyecto.

7.8. Usando la Sección 7.3 como directriz, calcule el SCT para OLCRS. Asegúrese de mostrar todo su trabajo. Seleccione un tipo de proyecto y cree un conjunto apropiado de tareas para el proyecto.

7.9. Defina una red de tareas para OLCRS o, alternativamente, para otro proyecto de software que le interese. Asegúrese de mostrar las tareas e hitos y adjuntar las estimaciones de esfuerzo y tiempo para cada tarea. Si es posible, utilice una herramienta de programación automática para realizar este trabajo.

7.10. Si dispone de una herramienta de programación automática, determine el camino crítico para la red definida en el problema 7.9.

7.11. Usando una herramienta de programación automática (si es posible) o papel y lápiz (si es necesario), desarrolle un gráfico de tiempo para el proyecto OLCRS.

7.12. Refine la tarea denominada **evaluación del riesgo tecnológico** en la sección 7.4 del mismo modo que se refinó el **ámbito del concepto** en la Sección 7.5.

7.13. Suponga que es un gestor de proyectos de software y que se le ha pedido que calcule estadísticas del valor ganado para un proyecto de software sencillo. El proyecto tiene 56 tareas planificadas que se estima que necesiten 582 personas-día para realizarlas. Al tiempo que ha sido solicitado para realizar el análisis del valor ganado, se han completado 12 tareas. Sin embargo, la planificación temporal del proyecto indica que se deberían haber completado 15 tareas. Están disponibles los siguientes datos de planificación (en personas-día):

tarea	esfuerzo planificado	esfuerzo real
1	12,0	12,5
2	15,0	11,0
3	13,0	17,0
4	8,0	9,5
5	9,5	9,0
6	18,0	19,0
7	10,0	10,0
8	4,0	4,5
9	12,0	10,0
10	6,0	6,5
11	5,0	4,0
12	14,0	14,5
13	16,0	-
14	6,0	-
15	8,0	-

Calcule IDP, la varianza de la planificación, el porcentaje planificado para terminación, el porcentaje completo IDC y la varianza del coste para el proyecto.

7.14. Es posible utilizar EED como una métrica para el seguimiento de errores en un proyecto de software. Estudie los pros y los contras de utilizar EED para este propósito.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

McConnel (*Rapid Development,* Microsoft Press, 1996) presenta un estudio excelente acerca de los aspectos que conducen a una planificación de proyectos de software demasiado optimista y lo que se puede hacer con ella. O'Connell (*How to Run Successful Projects II:The Silver Bullet,* Prentice Hall, 1997) presenta un enfoque paso a paso para la gestión de proyectos que pueden ayudarle a desarrollar una planificación temporal realista para sus proyectos.

Los aspectos de la planificación temporal están cubiertos en la mayoría de los libros sobre gestión de proyectos de software. McConnell (*Software Project Survival Guide,* Microsoft Press, 1998), Hoffman y Beaumont (*Application Development: Managing a Project's Life Cycle,* Midrange Computing, 1997), Wysoki y sus colegas (*Effective Project Management,* Wiley, 1995) y Whitten (*Managing Software Development Projects,* 2.ª ed., Wiley, 1995) consideran el tema en detalle. Boddie (*Crunch Mode,* Prentice-Hall, 1987) han escrito un libro para todos los gestores que «tienen 90 días para hacer un proyecto de seis meses».

También se puede obtener información que merece la pena en los libros de gestión de proyectos de propósito general. Entre ellos se encuentran:

Kezner, H., *Project Management: A Systems Approach to Planning, Scheduling, and Controlling, Wiley,* 1998.

Lewis, J. P., *Mastering Project Management: Applying Advanced Concepts of Systems Thinking, Control and Evaluation,* McGraw-Hill, 1988.

Fleming y Koppelman (*Earned Value Project Management,* Project Management Institute Publications, 1996) tratan con mucho detalle el uso de técnicas de valor ganado para el seguimiento y control de proyectos.

En Internet están disponibles una gran variedad de fuentes de información relacionadas con temas de gestión y planificación temporal de proyectos. Se puede encontrar una lista actualizada con referencias a sitios (páginas) web que son relevantes para la planificación en **http://www.pressman5.com.**

.

CAPÍTULO

8 GARANTÍA DE CALIDAD DEL SOFTWARE (SQA/GCS)*

EL enfoque de ingeniería del software descrito en este libro se dirige hacia un solo objetivo: producir software de alta calidad. Pero a muchos lectores les inquietará la pregunta: «¿Qué es la calidad del software?»

Philip Crosby [CR079], en su famoso libro sobre calidad, expone esta situación:

> El problema de la gestión de la calidad no es lo que la gente no sabe sobre ella. El problema es lo que creen que saben...
>
> A este respecto, la calidad tiene mucho en común con el sexo. Todo el mundo lo quiere. (Bajo ciertas condiciones, por supuesto.) Todo el mundo cree que lo conoce. (Incluso aunque no quiera explicarlo.) Todo el mundo piensa que su ejecución sólo es cuestión de seguir las inclinaciones naturales. (Después de todo, nos las arreglamos de alguna forma.) Y, por supuesto, la mayoría de la gente piensa que los problemas en estas áreas están producidos por otra gente. (Como si sólo ellos se tomaran el tiempo para hacer las cosas bien.)

Algunos desarrolladores de software continúan creyendo que la calidad del software es algo en lo que empiezan a preocuparse una vez que se ha generado el código. ¡Nada más lejos de la realidad! La garantía de calidad del software (SQA, Software Quality Assurance GCS, Gestión de calidad del software) es una actividad de protección (Capítulo 2) que se aplica a lo largo de todo el proceso del software. La **SQA** engloba: (1) un enfoque de gestión de calidad; (2) tecnología de ingeniería del software efectiva (métodos y herramientas); (3) revisiones técnicas formales que se aplican durante el proceso del software; (4) una estrategia de prueba multiescalada; (5) el control de la documentación del software y de los cambios realizados; (6) un procedimiento que asegure un ajuste a los estándares de desarrollo del software (cuando sea posible), y (7) mecanismos de medición y de generación de informes.

En este capítulo nos centraremos en los aspectos de gestión y en las actividades específicas del proceso que permitan a una organización de software asegurar que hace «las cosas correctas en el momento justo y de la forma correcta».

VISTAZO RÁPIDO

¿Qué es? No es suficiente hablar por hablar diciendo que la calidad del software es importante, tienes que: (1) definir explícitamente lo que significa «calidad del software»; (2) crear un conjunto de actividades que ayuden a garantizar que todo producto de la ingeniería del software presenta alta calidad; (3) llevar a cabo actividades de garantía de calidad en cada proyecto de software; 4) utilizar métricas para desarrollar estrategias que mejoren el proceso del software y, como consecuencia, mejoren la calidad del producto final.

¿Quién lo hace? Todo el que esté relacionado con el proceso de ingeniería del software es responsable de la calidad.

¿Por qué es importante? Lo puedes hacer correctamente o lo puedes repetir. Si un equipo de software aplica la calidad a todas las actividades de la ingeniería del software, reducirá la cantidad de trabajo repetido que deba realizar. Esto supondrá costes más bajos y, lo que es más importante, mejorará el tiempo de llegada al mercado.

¿Cuáles son los pasos? Antes de que se puedan iniciar las actividades de garantía de calidad del software, es importante definir la «calidad del software» a un número diferente de niveles de abstracción. Una vez que comprendes lo que es la calidad, un equipo de software debe identificar un conjunto de actividades de garantía de calidad del software que eliminarán los errores de los productos realizados antes de que ocurran.

¿Cuál es el producto obtenido? Para definir una estrategia de SQA para el equipo de software se ha creado un plan de garantía de calidad del software. Durante el análisis, diseño y codificación, el producto principal de SQA es un breve informe de la revisión técnica formal. Durante las pruebas, se realizan los planes y procedimientos de prueba. También se pueden generar otros productos de trabajo relacionados con la mejora del proceso.

¿Cómo puedo asegurar que lo he hecho correctamente? Encontrar los errores antes que pasen a ser defectos!. Esto es, trabajar para mejorar tu eficiencia en la eliminación de errores (Capítulos 4 y 7), reduciendo de este modo la cantidad de trabajo repetido que tiene que hacer tu equipo de software.

* N. del T.: En español, GCS. Se conservan las siglas SQA en inglés por estar muy extendido su uso para la jerga informática. A veces se suelen traducir estas siglas también por «Aseguramiento de la Calidad del Software».

8.1 CONCEPTOS DE CALIDAD[1]

Se dice que dos copos de nieve no son iguales. Ciertamente cuando se observa caer la nieve, es difícil imaginar que son totalmente diferentes, por no mencionar que cada copo posee una estructura única. Para observar las diferencias entre los copos de nieve, debemos examinar los especímenes muy de cerca, y quizá con un cristal de aumento. En efecto, cuanto más cerca los observemos, más diferencias podremos detectar.

Este fenómeno, *variación entre muestras*, se aplica a todos los productos del hombre así como a la creación natural. Por ejemplo, si dos tarjetas de circuito «idénticas» se examinan muy de cerca, podremos observar que las líneas de cobre sobre las tarjetas difieren ligeramente en la geometría, colocación y grosor. Además, la localización y el diámetro de los orificios de las tarjetas también varían.

Cita:
La gente olvida cómo de rápido hiciste un trabajo-pero siempre recuerda cómo de bien lo hiciste.
Howard Newton

El control de variación es el centro del control de calidad. Un fabricante quiere reducir la variación entre los productos que se fabrican, incluso cuando se realiza algo relativamente sencillo como la duplicación de disquetes. Seguramente, esto puede no ser un problema —la duplicación de disquetes es una operación de fabricación trivial y podemos garantizar que se crean duplicados exactos de software—.

¿Podemos?. Necesitamos asegurar que las pistas se sitúen dentro de una tolerancia específica para que la gran mayoría de las disqueteras puedan leer los disquetes. Además, necesitamos asegurar que el flujo magnético para distinguir un cero de un uno sea suficiente para que los detecten las cabezas de lectura/escritura. Las máquinas de duplicación de discos aceptan o rechazan la tolerancia. Por consiguiente, incluso un proceso «simple», como la duplicación, puede encontrarse con problemas debidos a la variación entre muestras.



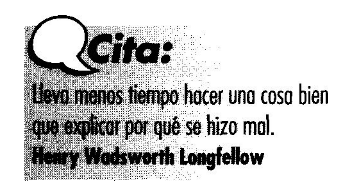
Controlar la variación es la clave de un producto de alta calidad. En el contexto del software, nos esforzamos en controlar la variación en el proceso que aplicamos, recursos que consumimos y los atributos de calidad del producto final.

¿Cómo se aplica esto al software? ¿Cómo puede una organización de desarrollo de software necesitar controlar la variación? De un proyecto a otro, queremos reducir la diferencia entre los recursos necesarios planificados para terminar un proyecto y los recursos reales utilizados, entre los que se incluyen personal, equipo y tiempo. En general, nos gustaría asegurarnos de que nuestro programa de pruebas abarca un porcentaje conocido del software de una entrega a otra. No sólo queremos reducir el número de defectos que se extraen para ese campo, sino también nos gustaría asegurarnos de que los errores ocultos también se reducen de una entrego a otra. (Es probable que nuestros clientes se molesten si la tercera entrega de un producto tiene diez veces más defectos que la anterior.) Nos gustaría reducir las diferencias en velocidad y precisión de nuestras respuestas de soporte a los problemas de los clientes. La lista se podría ampliar más y más.

8.1.1. Calidad

El *American Heritage Dictionary*, define la calidad como «una característica o atributo de algo». Como un atributo de un elemento, la calidad se refiere a las características mensurables —cosas que se pueden comparar con estándares conocidos como longitud, color, propiedades eléctricas, maleabilidad, etc.—. Sin embargo, el software en su gran extensión, como entidad intelectual, es más difícil de caracterizar que los objetos físicos.

No obstante, **sí** existen las medidas de características de un programa. Entre estas propiedades se incluyen complejidad ciclomática, cohesión, número de puntos de función, líneas de código y muchas otras estudiadas en los Capítulos 19 y **24.** Cuando se examina un elemento según sus características mensurables, se pueden encontrar dos tipos de calidad: calidad del diseño y calidad de concordancia.



La *calidad de diseño* se refiere a las características que especifican los ingenieros de software para un elemento. El grado de materiales, tolerancias y las especificaciones del rendimiento contribuyen a la calidad del diseño. Cuando se utilizan materiales de alto grado y se

[1] Esta sección, escrita por Michael Ctovcky, se ha adaptado de «Fundamentals of ICO 9000», un libro de trabajo desarrollado para Essential Software Engineering, un vídeo curriculum desarrollado por R.S. Pressman&Asociados, Inc. Reimpresión autorizada.

especifican tolerancias más estrictas y niveles más altos de rendimiento, la calidad de diseño de un producto aumenta, si el producto se fabrica de acuerdo con las especificaciones.

La *calidad de concordancia* es el grado de cumplimiento de las especificaciones de diseño durante su realización. Una vez más, cuanto mayor sea el grado de cumplimento, más alto será el nivel de calidad de concordancia.

En el desarrollo del software, la calidad de diseño comprende los requisitos, especificaciones y el diseño del sistema. La calidad de concordancia es un aspecto centrado principalmente en la implementación. Si la implementación sigue el diseño, y el sistema resultante cumple los objetivos de requisitos y de rendimiento, la calidad de concordancia es alta.

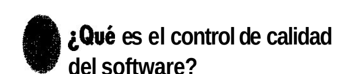
Pero, ¿son la calidad del diseño y la calidad de concordancia los Únicos aspectos que deben considerar los ingenieros de software? Robert Glass [GLA98] establece para ello una relacion más «intuitiva»:

> satisfacción del usuario = producto satisfactorio + buena calidad + entrega dentro de presupuesto y del tiempo establecidos

En la última línea, Glass afirma que la calidad es importante, pero si el usuario no queda satisfecho, ninguna otra cosa realmente importa. DeMarco [DEM99] refuerza este punto de vista cuando afirma: «La calidad del producto es una función de cuánto cambia el mundo para mejor.» Esta visión de la calidad establece que si el producto de software proporciona un beneficio sustancial a los usuarios finales, pueden estar dispuestos para tolerar problemas ocasionales del rendimiento o de fiabilidad.

8.1.2. Control de calidad

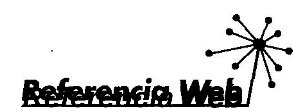
El control de cambios puede equipararse al control de calidad. Pero, ¿cómo se logra el control de calidad? ***El** control de calidad* es una serie de inspecciones, revisiones y pruebas utilizados a lo largo del proceso del software para asegurar que cada producto cumple con los requisitos que le han sido asignados. El control de calidad incluye un bucle de realimentación (feedback) del proceso que creó el producto. La combinación de medición y realimentación permite afinar el proceso cuando los productos de trabajo creados fallan al cumplir sus especificaciones. Este enfoque ve el control de calidad como parte del proceso de fabricación.



Las actividades de control de calidad pueden ser manuales, completamente automáticas o una combinación de herramientas automáticas e interacción humana. Un concepto clave del control de calidad es que se hayan definido todos los productos y las especificaciones mensurables en las que se puedan comparar los resultados de cada proceso. El bucle de realimentación es esencial para reducir los defectos producidos.

8.1.3. Garantía de calidad

La *garantia de calidad* consiste en la auditoría y las funciones de información de la gestión. El objetivo de la garantía de calidad es proporcionar la gestión para informar de los datos necesarios sobre la calidad del producto, por lo que se va adquiriendo una visión más profunda y segura de que la calidad del producto está cumpliendo sus objetivos. Por supuesto, si los datos proporcionados mediante la garantía de calidad identifican problemas, es responsabilidad de la gestión afrontar los problemas y aplicar los recursos necesarios para resolver aspectos de calidad.



Se puede encontrar una gran variedad de recursos de calidad del software en **www.qualitytree.com/links/links.htm**

8.1.4. Coste de calidad

El *coste de calidad* incluye todos los costes acarreados en la búsqueda de la calidad o en las actividades relacionadas en la obtención de la calidad. Se realizan estudios sobre el coste de calidad para proporcionar una línea base del coste actual de calidad, para identificar oportunidades de reducir este coste, y para proporcionar una base normalizada de comparación. La base de normalización siempre tiene un precio. Una vez que se han normalizado los costes de calidad sobre un precio base, tenemos los datos necesarios para evaluar el lugar en donde hay oportunidades de mejorar nuestros procesos. Es más, podemos evaluar cómo afectan los cambios en términos de dinero.

Los costes de calidad se pueden dividir en costes asociados con la prevención, la evaluación y los fallos. Entre ***los** costes de prevención* se incluyen:

- planificación de la calidad,
- revisiones técnicas formales,
- equipo de pruebas,
- formación.

¿Cuáles son los componentes del coste de calidad?

Entre ***los** costes de evaluación* se incluyen actividades para tener una visión más profunda de la condición del producto «la primera vez a través de» cada proceso. **A** continuación se incluyen algunos ejemplos de costes de evaluación:

- inspección en el proceso y entre procesos,
- calibrado y mantenimiento del equipo,
- pruebas.



No tengo miedo de incurrir en costes significativos de prevención. Esté segura de que su inversión le proporcionará un beneficio excelente.

Los *costes de fallos* son los costes que desaparecerían si no surgieran defectos antes del envío de un producto a los clientes. Estos costes se pueden subdividir en costes de fallos internos y costes de fallos externos. *Los internos* se producen cuando se detecta un error en el producto antes de **su** envío. Entre estos se incluyen:

- retrabajo (revisión),
- reparación,
- análisis de las modalidades de fallos.

Los *costes de fallos externos* son los que se asocian a los defectos encontrados una vez enviado el producto al cliente. A continuación se incluyen algunos ejemplos de costes de fallos externos:

- resolución de quejas,
- devolución y sustitución de productos,
- soporte de línea de ayuda,
- trabajo de garantía.

Como es de esperar, los costes relativos para encontrar y reparar un defecto aumentan dramáticamente a medida que se cambia de prevención a detección y desde el fallo interno al externo. La Figura 8.1, basada en datos recopilados por Boehm [BOE81], ilustra este fenómeno.



las pruebas son necesarias, pero también es una forma costosa de encontrar errores. Gaste el tiempo en encontrar errores al comienzo del proceso y podrá reducir significativamente los costes de pruebas y depuración.

Kaplan y sus colegas [KAP95] refuerzan las estadísticas de costes anteriores informando con datos anecdóticos basados en un trabajo realizado en las instalaciones de desarrollo de IBM en Rochester:

> Se han dedicado 7.053 horas inspeccionando 200.000 líneas de código con el resultado de **3.112** errores potenciales descubiertos. Dando por sentado un coste de programador de 40 dólares por hora, el coste de eliminar 3.1**12** defectos ha sido de 282.120 dólares, o aproximadamente unos 91 dólares por defecto.
>
> Compare estos números con el coste de eliminación de defectos una vez que el producto se ha enviado al cliente. Suponga que no ha habido inspecciones, pero que los programadores han sido muy cuidadosos y solamente se ha escapado un defecto por 1.000 líneas de código [significativamente mejor que la media en industrial] en el producto enviado. Eso significaría que se tendrían que corregir todavía *200* defectos en la casa del cliente o después de la entrega. **A** un coste estimado de 25.000 dólares por reparación de campo, el coste sena de 5 millones de dólares, o aproximadamente **18** veces más caro que el coste total del esfuerzo de prevención de defectos.

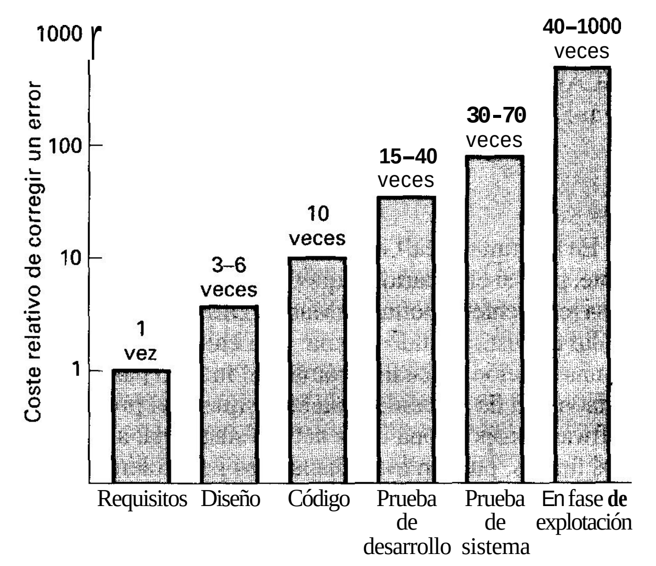


FIGURA 8.1. Coste relativo de corregir un error.

Es verdad que IBM produce software utilizado **por** cientos de miles de clientes y que sus costes por reparación en casa del cliente o después de la entrega pueden ser más altos que los de organizaciones de software que construyen sistemas personalizados. Esto, de ninguna manera, niega los resultados señalados anteriormente. Aunque la organización media de software tiene costes de reparación después de la entrega que son el 25 por 100 de los de IBM (¡la mayoría no tienen ni idea de cuales son sus costes!), se están imponiendo ahorros en el coste asociados con actividades de garantía y control de calidad.

8.2 LA TENDENCIA DE LA CALIDAD

Hoy en día **los** responsables expertos de compañías de todo el mundo industrializado reconocen que la alta calidad del producto se traduce en ahorro de coste y en una mejora general. Sin embargo, esto no era siempre el caso.

La tendencia de la calidad comenzó en los años cuarenta con el influyente trabajo de **W.** Edwards Deming [DEM86], y se hizo la primera verificación en Japón. Mediante las ideas de Deming como piedra angular, los

japoneses han desarrollado un enfoque sistemático para la eliminación de las causas raíz de defectos en productos. A lo largo de los años setenta y ochenta, su trabajo emigró al mundo occidental y a veces se llama «gestión total de calidad *(GTC)*»[2]. Aunque la terminología difiere según los diferentes países y autores, normalmente se encuentra una progresión básica de cuatro pasos que constituye el fundamento de cualquier programa de GTC.



Se puede aplicar GTC al software de computadora. El enfoque GTC se centra en la mejora continua del proceso.

El primer paso se llama *kuizen* y *se* refiere a un sistema de mejora continua del proceso. El objetivo de *kaizen* es desarrollar un proceso (en este caso, proceso del software) que sea visible, repetible y mensurable.

El segundo paso, invocado sólo una vez que se ha alcanzado *kuizen,* se llama *aturimae hinshitsu.* Este paso examina lo intangible que afecta al proceso y trabaja para optimizar su impacto en el proceso. Por ejemplo, el proceso de software se puede ver afectado por la alta rotación de personal que ya en sí mismo se ve afectado por reorganizaciones dentro de una compañía. Puede ser que una estructura organizativa estable haga mucho para mejorar la calidad del software. *Atarimae hinshitsu* llevaría a la gestión a sugerir cambios en la forma en que ocurre la reorganización.

Mientras que los dos primeros pasos se centran en el proceso, el paso siguiente llamado *kansei* (traducido como «los cinco sentidos») *se* centra en el usuario del producto (en este caso, software). En esencia, examinando la forma en que el usuario aplica el producto, *kunsei* conduce a la mejora en el producto mismo, y potencialmente al proceso que lo creó.

Finalmente, un paso llamado *miryokuteki hinshitsu* amplía la preocupación de la gestión **más** allá del producto inmediato. Este es un paso orientado a la gestión que busca la oportunidad en áreas relacionadas que se pueden identificar observando la utilización del producto en el mercado. En el mundo del software, *mirvokuteki hinshitsu* se podría ver como un intento de detectar productos nuevos **y** beneficiosos, o aplicaciones que sean una extensión de un sistema ya existente basado en computadora.

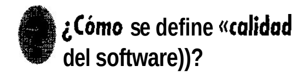


Se puede encontrar una gran variedad de recursos para la mejora continua del proceso y GTC en **deming.eng.clemsom.edu/**

Para la mayoría de las compañías, *kuizen* debería ser de preocupación inmediata. Hasta que se haya logrado un proceso de software avanzado (Capítulo 2), no hay muchos argumentos para seguir con los pasos siguientes.

8.3 GARANTÍA DE CALIDAD DEL SOFTWARE

Hasta el desarrollador de software más agobiado estará de acuerdo con que el software de alta calidad es una meta importante. Pero, ¿cómo definimos la calidad? Un bromista dijo una vez: «Cualquier programa hace algo bien, lo que puede pasar es que no sea lo que nosotros queremos que haga».



En los libros se han propuesto muchas definiciones ,&calidad del software. Por lo que a nosotros respecta, *la calidad del software* se define como:

> Concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos, con los estándares de desarrollo explícitamente documentados, y con las características implícitas que se espera de todo software desarrollado profesionalmente.

No hay duda de que la anterior definición puede ser modificada o ampliada. De hecho, no tendría fin una discusión sobre una definición formal de calidad del software. Para los propósitos de este libro, la anterior definición sirve para hacer hincapié en tres puntos importantes:

1. Los requisitos del software son la base de las medidas de la calidad. La falta de concordancia con los requisitos es una falta de calidad.
2. Los estándares especificados definen un conjunto de criterios de desarrollo que guían la forma en que se aplica la ingeniería del software. Si no se siguen esos criterios, casi siempre habrá falta de calidad.
3. Existe un conjunto de requisitos implícitos que a menudo no se mencionan (por ejemplo: el deseo por facilitar el uso y un buen mantenimiento). Si el software se ajusta a sus requisitos explícitos pero falla en alcanzar los requisitos implícitos, la calidad del software queda en entredicho.

8.3.1. Problemas de fondo

La historia de la garantía de calidad en el desarrollo de software es paralela a la historia de la calidad en la creación de hardware. Durante los primeros años de la infor-

[2] Consulte [ART92] para un estudio más completo del GTC y su uso en un contexto de software, y [KAP95] para un estudio sobre el uso de criterio «Balbridge Award» en el mundo del software.

mática (los años cincuenta y sesenta), la calidad era responsabilidad únicamente del programador. Durante los años setenta se introdujeron estándares de garantía de calidad para el software en los contratos militares para desarrollo de software y se han extendido rápidamente a los desarrollos de software en el mundo comercial [IEE94]. Ampliando la definición presentada anteriormente, la garantía de calidad del software (SQA) es un «patrón de acciones planificado y sistemático» [SCH97] que se requiere para asegurar la calidad del software. El ámbito de la responsabilidad de la garantía de calidad se puede caracterizar mejor parafraseando un popular anuncio de coches: «La calidad es la 1.ª tarea.» La implicación para el software es que muchos de los que constituyen una organización tienen responsabilidad de garantía de calidad del software —ingenieros de software, jefes de proyectos, clientes, vendedores, y aquellas personas que trabajan dentro de un grupo de SQA—.



Se puede encontrar un tutorial profundo y amplios recursos para la gestión de calidad en **www.management.gov**

El grupo de SQA sirve como representación del cliente en casa. Es decir, la gente que lleva a cabo'la SQA debe mirar el software desde el punto de vista del cliente. ¿Satisface de forma adecuada el software los factores de calidad apuntados en el Capítulo 19? ¿Se ha realizado el desarrollo del software de acuerdo con estándares preestablecidos? ¿Han desempeñado apropiadamente sus papeles las disciplinas técnicas como parte de la actividad de SQA? El grupo de SQA intenta responder a estas y otras preguntas para asegurar que se mantiene la calidad del software.

8.3.2. Actividades de SQA

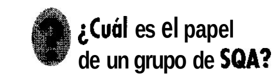
La garantía de calidad del software comprende una gran variedad de tareas, asociadas con dos constitutivos diferentes —los ingenieros de software que realizan trabajo técnico y un grupo de SQA que tiene la responsabilidad de la Planificación de garantía de calidad, supervisión, mantenimiento de registros, análisis e informes—.

Los ingenieros de software afrontan la calidad (y realizan garantía de calidad) aplicando métodos técnicos sólidos y medidas, realizando revisiones técnicas formales y llevando a cabo pruebas de software bien planificadas. Solamente las revisiones son tratadas en este capítulo. Los temas de tecnología se estudian en las Partes Tercera a Quinta de este libro.

Las reglas del grupo de SQA tratan de ayudar al equipo de ingeniería del software en la consecución de un producto final de alta calidad. El Instituto de Ingeniería del Software [PAU93] recomienda un conjunto de actividades de SQA que se enfrentan con la planificación de garantía de calidad, supervisión, mantenimiento de registros, análisis e informes. Éstas son las actividades que realizan (o facilitan) un grupo independiente de SQA:

Establecimiento de un plan de SQA para un proyecto. El plan se desarrolla durante la planificación del proyecto y es revisado por todas las partes interesadas. Las actividades de garantía de calidad realizadas por el equipo de ingeniería del software y el grupo SQA son gobernadas por el plan. El plan identifica:

- evaluaciones a realizar,
- auditorías y revisiones a realizar,
- estándares que se pueden aplicar al proyecto,
- procedimientos para información y seguimiento de errores,
- documentos producidos por el grupo SQA,
- realimentación de información proporcionada al equipo de proyecto del software.



Participación en el desarrollo de la descripción del proceso de software del proyecto. El equipo de ingeniería del software selecciona un proceso para el trabajo que se va a realizar. El grupo de SQA revisa la descripción del proceso para ajustarse a la política de la empresa, los estándares internos del software, los estándares impuestos externamente (por ejemplo: ISO 9001), y a otras partes del plan de proyecto del software.

Revisión de las actividades de ingeniería del software para verificar su ajuste al proceso de software definido. El grupo de SQA identifica, documenta y sigue la pista de las desviaciones desde el proceso y verifica que se han hecho las correcciones.

Auditoría de los productos de software designados para verificar el ajuste con los definidos como parte del proceso del software. El grupo de SQA revisa los productos seleccionados; identifica, documenta y sigue la pista de las desviaciones; verifica que se han hecho las correcciones, e informa periódicamente de los resultados de su trabajo al gestor del proyecto.

Asegurar que las desviaciones del trabajo y los productos del software se documentan y se manejan de acuerdo con un procedimiento establecido. Las desviaciones se pueden encontrar en el plan del proyecto, en la descripción del proceso, en los estándares aplicables o en los productos técnicos.

Registrar lo que no se ajuste a los requisitos e informar a sus superiores. Los elementos que no se ajustan a los requisitos están bajo seguimiento hasta que se resuelven.

Además de estas actividades, el grupo de SQA coordina el control y la gestión de cambios (Capítulo 9) y ayuda a recopilar y a analizar las métricas del software.

REVISIONES DEL SOFTWARE

Las revisiones del software son un «filtro» para el proceso de ingeniería del software. Esto es, las revisiones se aplican en varios momentos del desarrollo del software y sirven para detectar errores y defectos que puedan así ser eliminados. Las revisiones del software sirven para «purificar» las actividades de ingeniería del software que suceden como resultado del análisis, el diseño y la codificación. Freedman y Weinberg [FRE90] argumentan de la siguiente forma la necesidad de revisiones:

> El trabajo técnico necesita ser revisado por la misma razón que **los** lápices necesitan gomas: errar es humano. La segunda razón por la que necesitamos revisiones técnicas es que, aunque **la** gente es buena descubriendo algunos de **sus** propios errores, algunas clases de errores se le pasan por alto más fácilmente al que **los** origina que a otras personas. El proceso de revisión es, por tanto, la respuesta a la plegaria de Robert Bums:
>
> ¡Qué *gran* regalo sería poder vemos como nos ven **los** demás!
>
> Una revisión —cualquier revisión— es una forma de aprovechar la diversidad de un grupo de personas para:
>
> 1. señalar la necesidad de mejoras en el producto de una sola persona o un equipo;
> 2. confirmar las partes de un producto en las que no es necesaria o no es deseable una mejora; y
> 3. conseguir un trabajo técnico de una calidad más uniforme, o al menos más predecible, que la que puede ser conseguida sin revisiones, con el **fin** de hacer más manejable **el** trabajo técnico.



*Al **igual** que **los** filtros de **agua,** las **RTF's** tienden o retardar el «flujo» de las actividades de ingeniería del **software**. Muy pocos y el flujo es «sucio». Muchos y el flujo se reducirá o un goteo. Utilice métricos poro **determinar** qué revisiones son efectivas y cuáles no. **Saque** los que no **sean** efectivos fuero del flujo.*

Existen muchos tipos diferentes de revisiones que se pueden llevar adelante como parte de la ingeniería del software. Cada una tiene su lugar. Una reunión informal alrededor de la máquina de café es una forma de revisión, si se discuten problemas técnicos. Una presentación formal de un diseño de software a una audiencia de clientes, ejecutivos y personal técnico es una forma de revisión. Sin embargo, en este libro nos centraremos en la *revisión técnica formal (RTF)* —a veces denominada *inspección*—. Una revisión técnica formal es el filtro más efectivo desde el punto de vista de garantía de calidad. Llevada a cabo por ingenieros del software (y otros), la RTF es para ellos un medio efectivo para mejorar la calidad del software.

8.4.1. Impacto de los defectos del software sobre el coste

El *IEEE Standard Dictionary of Electrical and Electronics Terms* (IEEE Standard 100-1992) define un *defecto como* una «anomalía del producto». La definición de «fallo» en el contexto del hardware se puede encontrar en IEEE Standard 610. 12-1990:

> (a) Un defecto en un dispositivo de hardware o componente: por ejemplo, un corto circuito o un cable roto.
>
> (b) Un paso incorrecto, proceso o definición de datos en un programa de computadora. Nota: Esta definición se usa principalmente por la disciplina de tolerancia de fallos. En su uso normal, *los* términos «error» **y** «fallo» se utilizan para expresar este significado. Consulte también: fallo sensible al dato; fallo sensible al programa; fallos equivalentes; enmascaramiento de fallos; **fallo** intermitente.

Dentro del contexto del proceso del software, los términos defecto y fallo son sinónimos. Ambos implican un problema de calidad que es descubierto *después* de entregar el software a los usuarios finales (o a otra actividad del proceso del software). En los primeros capítulos, se utilizó el término *error* para representar un problema de calidad que es descubierto por los ingenieros del software (u otros) antes de entregar el software al usuario final (o a otra actividad del proceso del software).

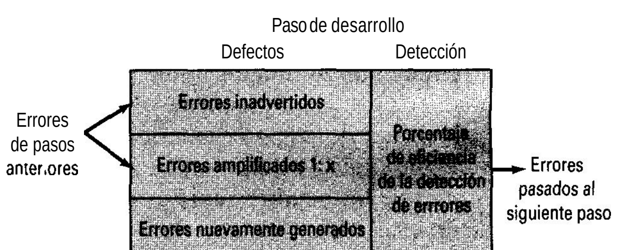


FIGURA 8.2. Modelo de amplificación de defectos.

El objetivo primario de las revisiones técnicas formales es encontrar errores durante el proceso, de forma que se conviertan en defectos después de la entrega del software. El beneficio obvio de estas revisiones técnicas formales es el descubrimiento de errores al principio para que no se propaguen al paso siguiente del proceso del software.



El objetivo principal de una RTF es encontrar errores antes de pasar a otra actividad de ingeniería del software o de entregar al cliente.

Una serie de estudios (TRW, Nippon Electric y Mitre Corp., entre otros) indican que las actividades del diseño introducen entre el 50 al 65 por ciento de todos los errores (y en último lugar, todos los defectos) durante

el proceso del software. Sin embargo, se ha demostrado que las revisiones técnicas formales son efectivas en un 75 por 100 a la hora de detectar errores [JON86]. Con la detección y la eliminación de un gran porcentaje de errores, el proceso de revisión reduce substancialmente el coste de los pasos siguientes en las fases de desarrollo y de mantenimiento.

Para ilustrar el impacto sobre el coste de la detección anticipada de errores, consideremos una serie de costes relativos que se basan en datos de coste realmente recogidos en grandes proyectos de software [IBM81][3]. Supongamos que un error descubierto durante el diseño cuesta corregirlo 1,0 unidad monetaria. De acuerdo con este coste, el mismo error descubierto justo antes de que comienza la prueba costará 6,5 unidades; durante la prueba 15 unidades; y después de la entrega, entre 60 y 100 unidades.

8.4.2. Amplificación y eliminación de defectos

Se puede usar un modelo de amplificación de defectos [IBM81] para ilustrar la generación y detección de errores durante los pasos de diseño preliminar, diseño detallado y codificación del proceso de ingeniería del software. En la Figura 8.2 se ilustra esquemáticamente el modelo. Cada cuadro representa un paso en el desarrollo del software. Durante cada paso se pueden generar errores que se pasan inadvertidos. La revisión puede fallar en descubrir nuevos errores y errores de pasos anteriores, produciendo un mayor número de errores que pasan inadvertidos. En algunos casos, los errores que pasan inadvertidos desde pasos anteriores se amplifican (factor de amplificación, X) con el trabajo actual. Las subdivisiones de los cuadros representan cada una de estas características y el porcentaje de eficiencia para la detección de errores, una función de la profundidad de la revisión.

La Figura 8.3 ilustra un ejemplo hipotético de la amplificación de defectos en un proceso de desarrollo de software en el que no se llevan a cabo revisiones. Como muestra la figura, se asume que cada paso descubre y corrige el 50 por 100 de los errores que le llegan, **sin** introducir ningún error nuevo (una suposición muy optimista). Antes de que comience la prueba, se han amplificado diez errores del diseño preliminar a 94 errores. **Al** terminar quedan 12 errores latentes. La Figura 8.4 considera las mismas condiciones, pero llevando a cabo revisiones del diseño y de la codificación dentro de cada paso del desarrollo. En este caso los 10 errores del diseño preliminar se amplifican a 24 antes del comienzo de la prueba. Sólo quedan ***3*** errores latentes. Recordando los costes relativos asociados con el descubrimiento y la corrección de errores, se puede establecer el coste total (con y sin revisiones para nuestro ejemplo hipotético). El número de errores encontrado durante cada paso mostrado en las figuras 8.3 y 8.4 se multiplica por el coste de eliminar un error (1.5 unidades de coste del diseño, 6.5 unidades de coste antes de las pruebas, 15 unidades de coste durante las pruebas, y 67 unidades de coste después de la entrega. Utilizando estos datos, se puede ver que el coste total para el desarrollo y el mantenimiento cuando se realizan revisiones es de 783 unidades. Cuando no hay revisiones, el coste total es de 2.177 unidades —casi tres veces más caro—.

Cita:

Algunas enfermedades, como dicen los médicos, en su comienzo son fáciles de curar pero difíciles de reconocer... pero con el paso del tiempo, cuando no se han reconocido y tratado al principio, se vuelven fáciles de reconocer pero difíciles de curar.

Niccolo Machiavelli

Para llevar a cabo revisiones, el equipo de desarrollo debe dedicar tiempo y esfuerzo, y la organización de desarrollo debe gastar dinero. Sin embargo, los resultados del ejemplo anterior no dejan duda de que hemos encontrado el síndrome de «pague ahora o pague, después, mucho más». Las revisiones técnicas formales (para el diseño y otras actividades técnicas) producen un beneficio en coste demostrable. Deben llevarse a cabo.

8.5 REVISIONES TÉCNICAS FORMALES

Una revisión técnica formal (RTF) es una actividad de garantía de calidad del software llevada a cabo por los ingenieros del software (y otros). Los objetivos de la **RTF** son: (1) descubrir errores en la función, la lógica o la implementación de cualquier representación del software; (2) verificar que el software bajo revisión alcanza sus requisitos; *(3)* garantizar que el software ha sido representado de acuerdo con ciertos estándares predefinidos; **(4)** conseguir un software desarrollado de forma uniforme y (5) hacer que los proyectos sean más manejables. Además, la **RTF** sirve como campo de entrenamiento, permitiendo que los ingenieros más jóvenes puedan observar los diferentes enfoques de análisis, diseño e implementación del software. La **RTF** también sirve para promover la seguridad y la continuidad, ya que varias personas se familiarizarán con partes del software que, de otro modo, no hubieran visto nunca.

[3] Aunque estos datos son de hace más de 20 años, pueden ser aplicados en un contexto moderno.

La RTF es realmente una clase de revisión que incluye recorridos, inspecciones, revisiones cíclicas y otro pequeño grupo de evaluaciones técnicas del software. Cada RTF se lleva a cabo mediante una reunión y sólo tendrá éxito si es bien planificada, controlada y atendida. En las siguientes secciones se presentan directrices similares a las de las inspecciones [FRE90, GIL93] como representativas para las revisiones técnicas formales.

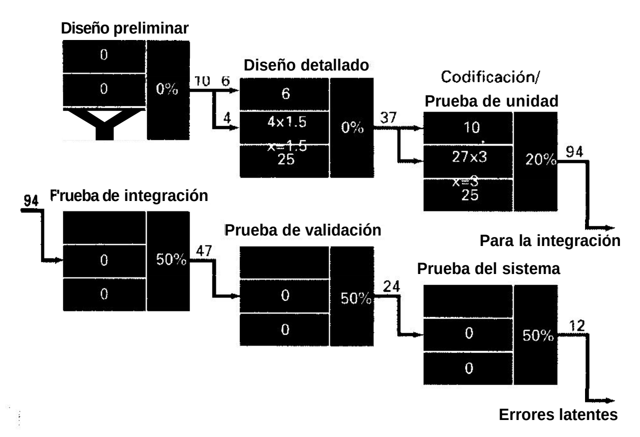


FIGURA 8.3. Amplificación de defectos, sin revisiones.

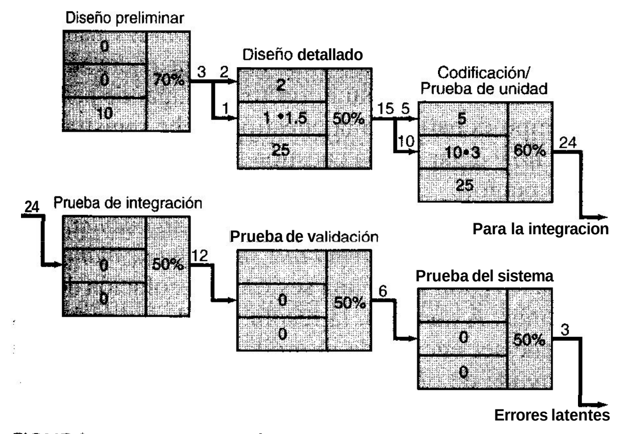
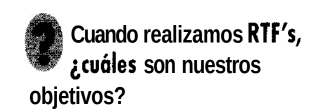


FIGURA 8.4. Amplificación de defectos, llevando a cabo revisiones.

8.5.1. La reunión de revisión

Independientemente del formato que se elija para la RTF, cualquier reunión de revisión debe acogerse a las siguientes restricciones:

- deben convocarse para la revisión (normalmente) entre tres y cinco personas;
- se debe preparar por adelantado, pero sin que requiera **más** de dos horas de trabajo a cada persona; y
- la duración de la reunión de revisión debe ser menor de dos horas.





Con las anteriores limitaciones, debe resultar obvio que cada RTF se centra en una parte específica (y pequeña) del software total. Por ejemplo, en lugar de intentar revisar un diseño completo, se hacen inspecciones para cada módulo (componente) o pequeño grupo de módulos. Al limitar el centro de atención de la RTF, la probabilidad de descubrir errores es mayor.

El centro de atención de la RTF es un producto de trabajo (por ejemplo, una porción de una especificación de requisitos, un diseño detallado del módulo, un listado del código fuente de un módulo). El individuo que ha desarrollado el producto *—el productor—* informa al jefe del proyecto de que el producto está terminado y que se requiere una revisión. El jefe del proyecto contacta con un *jefe de revisión,* que evalúa la disponibilidad del producto, genera copias del material del producto y las distribuye a dos o tres revisores para que se preparen por adelantado. Cada revisor estará entre una y dos horas revisando el producto, tomando notas y también familiarizándose con el trabajo. De forma concurrente, también el jefe de revisión revisa el producto y establece una agenda para la reunión de revisión que, normalmente, queda convocada para el día siguiente.

C VE

la **RTF** se centra en una porción relativamente pequeña de un producto de **trabajo.**

La reunión de revisión es llevada a cabo por el jefe de revisión, los revisores y el productor. Uno de **los** revisores toma el papel de registrador, o sea, de persona que registra (de forma escrita) todos los sucesos importantes que *se* produzcan durante la revisión. La RTF comienza con una explicación de la agenda y una breve introducción a cargo del productor. Entonces el productor procede con el «recorrido de inspección» del producto, explicando el material, mientras que los revisores exponen sus pegas basándose en su preparación previa. Cuando se descubren problemas o errores válidos, el registrador los va anotando.



Puede descargarse el **NASA SATC** *Formal Inspection Guideboock* de **satc.gsfc.nasa.gov/fi/fipage.html**

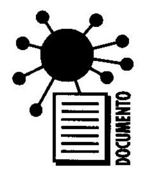
Al final de la revisión, todos los participantes en la **RTF** deben decidir si (1) aceptan el producto sin posteriores modificaciones; (2) rechazan el producto debido a los serios errores encontrados (una vez corregidos, ha de hacerse otra revisión) o *(3)* aceptan el producto provisionalmente (se han encontrado errores menores que deben ser corregidos, pero sin necesidad de otra posterior revisión). Una vez tomada la decisión, todos los participantes terminan firmando, indicando así que han participado en la revisión y que están de acuerdo con las conclusiones del equipo de revisión.

8.5.2. Registro e informe de la revisión

Durante la **RTF,** uno de los revisores (el registrador) procede a registrar todas las pegas que vayan surgiendo. Al final de la reunión de revisión, resume todas las pegas y genera una lista de sucesos de revisión. Además, prepara un informe sumario de la revisión técnica formal. El informe sumario de revisión responde a tres preguntas:

1. ¿Qué fue revisado?
2. ¿Quién lo revisó?
3. ¿Qué se descubrió y cuáles son las conclusiones?

El informe sumario de revisión es una página simple (con posibles suplementos). Se adjunta al registro histórico del proyecto y puede ser enviada al jefe del proyecto y a otras partes interesadas.



Informe Sumarial y lista de Sucesos de Revisión Técnica

La lista de sucesos de revisión sirve para dos propósitos: (1) identificar áreas problemáticas dentro del producto y (2) servir como lista de comprobación de puntos de acción que guíe al productor para hacer las correcciones. Normalmente se adjunta una lista de conclusiones al informe sumario.

Es importante establecer un procedimiento de seguimiento que asegure que los puntos de la lista de sucesos son corregidos adecuadamente. A menos que se haga así, es posible que las pegas surgidas «caigan en saco roto». Un enfoque consiste en asignar la responsabilidad del seguimiento al revisor jefe

8.5.3. Directrices para la revisión

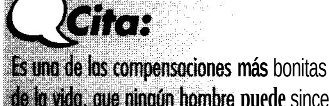
Se deben establecer de antemano directrices para conducir las revisiones técnicas formales, distribuyéndolas después entre los revisores, para ser consensuadas y, finalmente, seguidas. **A** menudo, una revisión incontrolada puede ser peor que no hacer ningún tipo de revisión. A continuación se muestra un conjunto mínimo de directrices para las revisiones técnicas formales:

1. *Revisar el producto, no al productor.* Una **RTF** involucra gente y egos. Conducida adecuadamente, la **RTF** debe llevar a todos los participantes a un sentimiento agradable de estar cumpliendo **su** deber. Si se lleva a cabo incorrectamente, la **RTF** puede tomar el aura de inquisición. Se deben señalar los errores educadamente; el tono de la reunión debe ser distendido y constructivo; no debe intentarse dificultar o batallar. El jefe de revisión debe moderar la reunión para garantizar que se mantiene un tono y una actitud adecuados y debe inmediatamente cortar cualquier revisión que haya escapado al control.



No señale bruscamente los errores. Una forma de ser educada es hacer una pregunto que permita al productor descubrir su propia errar.

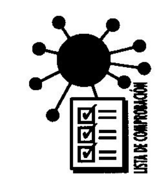
2. *Fijar una agenda y mantenerla.* Un mal de las reuniones de todo tipo es la *deriva.* La **RTF** debe seguir un plan de trabajo concreto. El jefe de revisión es el que carga con la responsabilidad de mantener el **plan** de la reunión y no debe tener miedo de cortar a la gente cuando se empiece a divagar.
3. *Limitar el debate y las impugnaciones.* Cuando un revisor ponga de manifiesto una pega, podrá no haber unanimidad sobre **su** impacto. En lugar de perder el tiempo debatiendo la cuestión, debe registrarse el hecho y dejar que la cuestión se lleve a cabo en otro momento.
4. *Enunciar áreas de problemas, pero no intentar resolver cualquier problema que se ponga de manifiesto.* Una revisión no es una sesión de resolución de problemas. A menudo, la resolución de los problemas puede ser encargada al productor por **sí** solo o con la ayuda de otra persona. La resolución de los problemas debe ser pospuesta para después de la reunión de revisión.



5. *Tomar notas escritas.* A veces es buena idea que el registrador tome las notas en una pizarra, de forma que las declaraciones o la asignación de prioridades pueda ser comprobada por el resto de **los** revisores, a medida que se va registrando la información.
6. *Limitar el número de participantes e insistir en la preparación anticipada.* Dos personas son mejores que una, pero catorce no son necesariamente mejores que cuatro. Se ha de mantener el número de participantes en el mínimo necesario. Además. todos los miembros

del equipo de revisión deben prepararse por anticipado. El jefe de revisión debe solicitar comentarios (que muestren que cada revisor ha revisado el material).

7. *Desarrollar una lista de comprobación para cada producto que haya de ser revisado,* Una lista de comprobaciones ayuda al jefe de revisión a estructurar la reunión de RTF y ayuda a cada revisor a centrarse en los asuntos importantes. Se deben desarrollar listas de comprobaciones para los documentos de análisis, de diseño, de codificación e incluso de prueba.



listas de comprobación de **RTF**

8. *Disponer recursos y una agenda para las RTF.* Para que las revisiones sean efectivas, se deben planificar como una tarea del proceso de ingeniería del software. Además se debe trazar un plan de actuación para las modificaciones inevitables que aparecen como resultado de una RTF.

9. *Llevar a cabo un buen entrenamiento de todos los revisores.* Por razones de efectividad, todos los participantes en la revisión deben recibir algún entrenamiento formal. El entrenamiento se debe basar en los aspectos relacionados con el proceso, así como las consideraciones de psicología humana que atañen a la revisión. Freedman y Weinberg [FRE90] estiman en un mes la curva de aprendizaje para cada veinte personas que vayan a participar de forma efectiva en las revisiones.

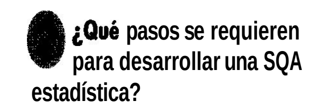
10. *Repasar las revisiones anteriores.* Las sesiones de información pueden ser beneficiosas para descubrir problemas en el propio proceso de revisión. El primer producto que se haya revisado puede establecer las propias directivas de revisión.

Como existen muchas variables (por ejemplo, número de participantes, tipo de productos de trabajo, tiempo y duración, enfoque de revisión específico) que tienen impacto en una revisión satisfactoria, una organización de software debería determinar qué método funciona mejor en un contexto local. Porter y sus colegas [POR95] proporcionan una guía excelente para este tipo de experimentos.

8 GARANTÍA DE CALIDAD ESTADÍSTICA

La garantía de calidad estadística refleja una tendencia, creciente en toda la industria, a establecer la calidad **más** cuantitativamente. Para el software, la garantía de calidad estadística implica los siguientes **pasos:**

1. Se agrupa y se clasifica la información sobre los defectos del software.
2. Se intenta encontrar la causa subyacente de cada defecto (por ejemplo, no concordancia con la especificación, error de diseño, incumplimiento de los estándares, pobre comunicación con el cliente).



¿Qué pasos se requieren para desarrollar una SQA estadística?

3. Mediante el principio de Pareto (el 80 por 100 de los defectos se pueden encontrar en el 20 por 100 de todas las posibles causas), se aísla el 20 por 100 (los «pocos vitales»).
4. Una vez que se han identificado los defectos vitales, se actúa para corregir los problemas que han producido los defectos.

Este concepto relativamente sencillo representa un paso importante hacia la creación de un proceso de ingeniería del software adaptativo en el cual se realizan cambios para mejorar aquellos elementos del proceso que introducen errores.

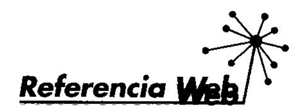


El 20 por 100 del código tiene el 80 por 100 de los defectos. ¡Encuéntrelos, corríjalos!.
Lowell Arthur

Para ilustrar el proceso, supongamos que una organización de desarrollo de software recoge información sobre defectos durante un período de un año. Algunos de los defectos se descubren mientras se desarrolla el software. Otros se encuentran después de que el software se haya distribuido al usuario final. Aunque se descubren cientos de errores diferentes, todos se pueden encontrar en una (o más) de las siguientes causas:

- Especificación incompleta o errónea (EIE).
- Mala interpretación de la comunicación del cliente (MCC).
- Desviación deliberada de la especificación (DDE).
- Incumplimiento de los estándares de programación (IEP).
- Error en la representación de los datos (ERD).
- Interfaz de módulo inconsistente (IMI).
- Error en la lógica de diseño (ELD).
- Prueba incompleta o errónea (PIE).
- Documentación imprecisa o incompleta (DII).
- Error en la traducción del diseño al lenguaje de programación (TLP).

- Interfaz hombre-máquina ambigua o inconsistente (IHM).
- Varios (VAR).



La Asociación China de Calidad del Software presenta uno de los sitios web más completos para la calidad en **www.cosq.org**

Para aplicar la SQA estadística se construye la Tabla 8.1. La tabla indica que EIE, MCC y ERD son las *causas vitales* que contabilizan el 53 por 100 de todos los errores. Sin embargo, debe observarse que si sólo se consideraran errores serios, se seleccionarían las siguientes causas vitales: EIE, ERD, TLP y ELD. Una vez determinadas las causas vitales, la organización de desarrollo de software puede comenzar la acción correctiva. Por ejemplo, para corregir la MCC, el equipo de desarrollo del software podría implementar técnicas que facilitaran la especificación de la aplicación (Capítulo 11) para mejorar la calidad de la especificación y la comunicación con el cliente. Para mejorar el ERD, el equipo de desarrollo del software podría adquirir herramientas CASE para la modelización de datos y realizar revisiones del diseño de datos más rigurosas.

Es importante destacar que la acción correctiva se centra principalmente en las causas vitales. Cuando éstas son corregidas, nuevas candidatas saltan al principio de la lista.

Se han mostrado las técnicas de garantía de calidad del software estadísticas para proporcionar una mejora sustancial en la calidad [ART97]. En algunos casos, las organizaciones de software han conseguido una reducción anual del 50 por 100 de los errores después de la aplicación de estas técnicas.

Junto con la recopilación de información sobre defectos, los equipos de desarrollo del software pueden calcular un *indice de errores* (IE) para cada etapa principal del proceso de ingeniería del software [IEE94]. Después del análisis, el diseño, la codificación, la prueba y la entrega, se recopilan los siguientes datos:

E_i = número total de defectos descubiertos durante la etapa i-ésima del proceso de ingeniería del software;
S_i = número de defectos graves;
M_i = número de defectos moderados;
T_i = número de defectos leves;
PS = tamaño del producto (LDC, sentencias de diseño, páginas de documentación) en la etapa i-ésima.
W_s, W_m, W_t = factores de peso de errores graves, moderados, y leves, en donde los valores recomendados son $W_s = 10$, $W_m = 3$, $W_t = 1$. Los factores de peso de cada fase deberían agrandarse a medida que el desarrollo evoluciona. Esto favorece a la organización que encuentra los errores al principio.

En cada etapa del proceso de ingeniería del software se calcula *un índice de fase*, IF_i :

$$IF_i = W_s (S_i/E_i) + W_m (M_i/E_i) + W_t (T_i/E_i)$$

El *indice de errores* (IE) se obtiene mediante el cálculo del defecto acumulativo de cada IF*i*, asignando más peso a los errores encontrados más tarde en el proceso de ingeniería del software, que a los que se encuentran en las primeras etapas:

$$IE = \sum (i \times IF_i) / PS$$
$$= (IF_1 + 2IF_2 + 3IF_3 + \ldots iIF_i) / PS$$

	Total		Grave		Moderado		Leve	
Error	**No.**	**%**	**No.**	**%**	**No.**	**%**	**No.**	**%**
IEE	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
DDE	48	5%	1	1%	24	6%	23	5%
IEP	25	3%	0	0%	15	4%	10	2%
ERD	130	14%	26	20%	68	18%	36	8%
IMI	58	6%	9	7%	18	5%	31	7%
ELD	45	5%	14	11%	12	3%	19	4%
PIE	95	10%	12	9%	35	9%	48	11%
DII	36	4%	2	2%	20	5%	14	3%
TLP	60	6%	15	12%	19	5%	26	6%
IHM	28	3%	3	2%	17	4%	8	2%
VAR	56	6%	0	0%	15	4%	41	9%
Totales	942	100%	128	100%	379	100%	435	100%

TABLA 8.1. Recolección de datos para la SQA estadística

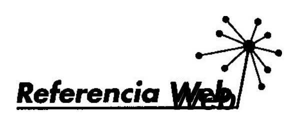
Se puede utilizar el índice de errores junto con la información recogida en la Tabla 8.1, para desarrollar una indicación global de la mejora en la calidad del software.

La aplicación de la SQA estadística y el principio de Pareto se pueden resumir en una sola frase: *¡Utilizar el tiempo para centrarse en cosas que realmente interesan, pero primero asegurarse que se entiende qué es lo que realmente interesa!*

Un estudio exhaustivo de la SQA estadística se sale del alcance de este libro. Los lectores interesados deberían consultar [SCH98], [KAP95] y [KAN95].

FIABILIDAD DEL SOFTWARE

No hay duda de que la fiabilidad de un programa de computadora es un elemento importante de su calidad general. Si un programa falla frecuente y repetidamente en su funcionamiento, no importa si el resto de los factores de calidad son aceptables.



El Centro de Análisis de Fiabilidad proporciona mucha información útil sobre la fiabilidad, mantenibilidad, soporte y calidad en **rac.iitri.org**

La fiabilidad del software, a diferencia de otros factores de calidad, puede ser medida o estimada mediante datos históricos o de desarrollo. La ***fiabilidad del*** software se define en términos estadísticos como «la probabilidad de operación libre de fallos de un programa de computadora en un entorno determinado y durante un tiempo específico» [MUS87]. Por ejemplo, un programa X puede tener una fiabilidad estimada de 0,96 durante un intervalo de proceso de ocho horas. En otras palabras, si se fuera a ejecutar el programa X 100 veces, necesitando ocho horas de tiempo de proceso (tiempo de ejecución), lo probable es que funcione correctamente (sin fallos) 96 de cada 100 veces.

Siempre que se habla de fiabilidad del software, surge la pregunta fundamental: ¿Qué se entiende por el término fallo? En el contexto de cualquier discusión sobre calidad y fiabilidad del software, el fallo es cualquier falta de concordancia con los requisitos del software. Incluso en esta definición existen grados. Los fallos pueden ser simplemente desconcertantes o ser catastróficos. Puede que un fallo sea corregido en segundos mientras que otro lleve semanas o incluso meses. Para complicar más las cosas, la corrección de un fallo puede llevar a la introducción de otros errores que, finalmente, lleven a más fallos.

8.7.1. Medidas de fiabilidad y de disponibilidad

Los primeros trabajos sobre fiabilidad intentaron extrapolar las matemáticas de la teoría de fiabilidad del hardware (por ejemplo: [ALV64]) a la predicción de la fiabilidad del software. La mayoría de los modelos de fiabilidad relativos al hardware van más orientados a los fallos debidos al desajuste que a los fallos debidos a defectos de diseño. En el hardware, son más probables los fallos debidos al desgaste físico (por ejemplo: el efecto de la temperatura, de la corrosión y los golpes) que los fallos relativos al diseño. Desgraciadamente, para el software lo que ocurre es lo contrario. De hecho, todos los fallos del software, se producen por problemas de diseño o de implementación; el desajuste (consulte el Capítulo 1) no entra en este panorama.



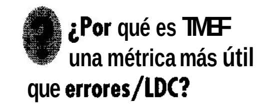
Los problemas de la fiabilidad del software se deben casi siempre a errores en el diseño o en la implementación.

Todavía se debate sobre la relación entre los conceptos clave de la fiabilidad del hardware y su aplicabilidad al software (por ejemplo, [LIT89], [ROO90]). Aunque aún falta por establecer un nexo irrefutable, merece la pena considerar unos cuantos conceptos que conciernen a ambos elementos de los sistemas.

Considerando un sistema basado en computadora, una sencilla medida de la fiabilidad es el tiempo medio ***entre*** *fallos (TMEF)*, donde;

$$TMEF = TMDF + TMDR$$

Las siglas TMDF y TMDR corresponden a tiempo medio de fallo y tiempo medio de reparación, respectivamente.



Muchos investigadores argumentan que el TMDF es, con mucho, una medida más Útil que los defectos/KLDC o defectos/PF. Sencillamente, el usuario final se enfrenta a los fallos, no al número total de errores. Como cada error de un programa no tiene la misma tasa de fallo, la cuenta total de errores no es una buena indicación de la fiabilidad de un sistema. Por ejemplo, consideremos un programa que ha estado funcionando durante 14 meses. Muchos de los errores del programa pueden pasar desapercibidos durante décadas antes de que se detecten. El TMEF de esos errores puede ser de 50 e incluso de 100 años. Otros errores, aunque no se hayan descubierto aún, pueden tener una tasa de fallo de 18 ó 24 meses. Incluso aunque se eliminen todos los errores de la primera categoría (los que tienen un gran TMEF), el impacto sobre la fiabilidad del software será muy escaso.

Además de una medida de la fiabilidad debemos obtener una medida de la disponibilidad. La *disponibilidad del software* es la probabilidad de que un programa funcione de acuerdo con los requisitos en un momento dado, y se define como:

$$Disponibilidad = [TMDF/(TMDF + TMDR)] \times 100\%$$

La medida de fiabilidad TMEF es igualmente sensible al TMDF que al TMDR. La medida de disponibilidad es algo más sensible al TMDR, una medida indirecta de la facilidad de mantenimiento del software.

8.7.2. Seguridad del software

Leveson [LEV86] discute el impacto del software en sistemas críticos de seguridad, diciendo:

> Antes de que se usara software en sistemas críticos de seguridad, normalmente éstos se controlaban mediante dispositivos electrónicos y mecánicos convencionales (no programables). Las técnicas de seguridad de sistemas se diseñaban para hacer frente a fallos aleatorios en esos dispositivos [no programables]. Los errores humanos de diseño no se consideraban porque se suponía que todos los defectos producidos por los errores humanos se podían evitar o eliminar completamente antes de su distribución y funcionamiento.



Cita:
No puedo imaginar cualquier condición que podría causar el hundimiento de este barco. La construcción naval moderna ha ido más allá de eso.
E. I. Smith, Capitan del Titanic

Cuando se utiliza el software como parte del sistema de control, la complejidad puede aumentar en un orden de magnitud o más. Los defectos sutiles de diseño, producidos por un error humano —algo que se puede descubrir y eliminar en el control convencional basado en el hardware— llegan a ser mucho más difíciles de descubrir cuando se utiliza el software.

La *seguridad del software* es una actividad de garantía de calidad del software que se centra en la identificación y evaluación de los riesgos potenciales que pueden producir un impacto negativo en el software y hacer que falle el sistema completo. Si se pueden identificar pronto los riesgos en el proceso de ingeniería del software podrán especificarse las características del diseño del software que permitan eliminar o controlar los riesgos potenciales.

Como parte de la seguridad del software, se puede dirigir un proceso de análisis y modelado. Inicialmente, se identifican los riesgos y se clasifican por su importancia y su grado de riesgo. Por ejemplo, algunos de los riesgos asociados con el control basado en computadora del sistema de conducción de un automóvil podrían ser:

- Produce una aceleración incontrolada que no se puede detener.
- No responde a la presión del pedal del freno (deteniéndose).
- No responde cuando se activa el contacto.
- Pierde o gana velocidad lentamente.

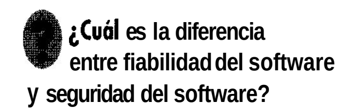
Cuando se han identificado estos riesgos del sistema, se utilizan técnicas de análisis para asignar su gravedad y su probabilidad de ocurrencia[4]. Para que sea efectivo, se tiene que analizar el software en el contexto del sistema completo. Por ejemplo, puede que un sutil error en la entrada del usuario (las personas son componentes del sistema) se magnifique por un fallo del software que producen los datos de control que actúan de forma inadecuada sobre un dispositivo mecánico. Si se dan un conjunto de condiciones externas del entorno (y sólo si se dan), la situación inadecuada del dispositivo mecánico producirá un fallo desastroso. Se pueden usar técnicas de análisis, como el *análisis del árbol de fallos* [VES81], la *lógica de tiempo real* [JAN86] *o los modelos de redes de Petri* [LEV87], para predecir la cadena de sucesos que pueden producir los riesgos y la probabilidad de que se dé cada uno de los sucesos que componen la cadena.



Referencia Web

Se pueden encontrar documentos sobre la seguridad del software (y un glosario detallado) que merecen la peno en **www.rstcorp.com/hotlist/topics-safety.html**

Una vez que se han identificado y analizado los riesgos, se pueden especificar los requisitos relacionados con la seguridad para el software. Esto es, la especificación puede contener una lista de eventos no deseables y las respuestas del sistema a estos eventos. El papel del software en la gestión de eventos no deseados es entonces apropiado.



¿Cuál es la diferencia entre fiabilidad del software y seguridad del software?

Aunque la fiabilidad y la seguridad del software están bastante relacionadas, es importante entender la sutil diferencia que existe entre ellas. La fiabilidad del software utiliza el análisis estadístico para determinar la probabilidad de que pueda ocurrir un fallo del software. Sin embargo, la ocurrencia de un fallo no lleva necesariamente a un riesgo o a un accidente. La seguridad del software examina los modos según los cuales los fallos producen condiciones que pueden llevar a accidentes. Es decir, los fallos no se consideran en vacío, sino que se evalúan en el contexto de un completo sistema basado en computadora.

Un estudio completo sobre seguridad del software y análisis del riesgo va más allá del ámbito de este libro. Para aquellos lectores que estén más interesados, deberían consultar el libro de Leveson [LEV95] sobre este tema.

[4] Este método es análogo al método de análisis de riesgos descrito para la gestión del proyecto de software en el capítulo 6. La principal diferencia es el énfasis en aspectos de tecnología frente a temas relacionados con el proyecto.

8.8 PRUEBA DE ERRORES PARA SOFTWARE

Si William Shakespeare hubiera escrito sobre las condiciones del ingeniero de software moderno, podría perfectamente haber escrito: «Errar es humano, encontrar el error rápida y correctamente es divino.» En los años sesenta, un ingeniero industrial japonés, Shigeo Shingo [SHI86], que trabajaba en Toyota, desarrolló una técnica de garantía de calidad que conducía a la prevención y/o a la temprana corrección de errores en el proceso de fabricación. Denominado *poku-yoke* (prueba de errores), el concepto de Shingo hacía uso de dispositivos *poku-yoke* — mecanismos que conducen (1) a la prevención de un problema de calidad potencial antes de que éste ocurra, o (2) a la rápida detección de problemas de calidad si se han introducido ya—. Nosotros encontramos dispositivos *poka-yoke* en nuestra vida cotidiana (incluso si nosotros no tenemos conciencia de este concepto). Por ejemplo, el interruptor de arranque de un coche no trabaja si está metida una marcha en la transmisión automática (un *dispositivo de prevención);* un pitido de aviso del coche sonará si los cinturones de seguridad no están bien sujetos (un *dispositivo de detección de fallos).*

Un dispositivo de *poku-yoke* presenta un conjunto de características comunes:

- Es simple y barato —si un dispositivo es demasiado complicado y caro, no será efectivo en cuanto a costo—;
- Es parte del proceso - esto es, el dispositivo *poka-yoke* está integrado en una actividad de ingeniería—;
- Está localizado cerca de la tarea del proceso donde están ocurriendo los errores —por consiguiente proporciona una realimentación rápida en cuanto a la corrección de errores se refiere—.



Referencia Web

Se puede obtener una colección completa de recursos de *poka-yoke* en
www.campbell.berry.edu/faculty/igrout/pokayoke.shtml

Aunque el *poka-yoke* fue originariamente desarrollado para su uso en «control de calidad cero» [SHI86] para el hardware fabricado, puede ser adaptado para su uso en ingeniería del software. Para ilustrar esto, consideremos el problema siguiente [ROB97]:

> Una compañía de productos de software vende el software de aplicación en el mercado internacional. Los menús desplegables y todos los códigos asociados proporcionados con cada aplicación deben reflejar el lenguaje que se emplea en el lugar donde se usa. Por ejemplo, el elemento del menú del lenguaje inglés para «Close», tiene el mnemónico «C» asociado con ello. Cuando la aplicación se vende en un país de habla francesa, el mismo elemento del menú es «Fermer» con el mnemónico «F». Para llevar a cabo la entrada adecuada del menú para cada aplicación, un «localizador» (una persona que habla en el idioma local y con la terminología de ese lugar) traduce los menús de acuerda con el idioma en uso. El problema es asegurar que (1) cada entrada de menú (pueden existir cientos de ellas) se adecue a **los** estándares apropiados y que no existan conflictos, independientemente del lenguaje que se está utilizando.

El uso del *poka-yoke* para la prueba de diversos menús de aplicación desarrollados en diferentes lenguajes, tal **y** como se ha descrito anteriormente, se discute en un artículo de Harry Robinson [ROB97]:

> Nosotros primero decidimos dividir el problema de pruebas de menús en partes que puedan ser resueltas más fácilmente. Nuestro primer avance para la resolución del problema fue el comprender que existían dos aspectos separados para los catálogos de mensaje. Había por una parte el aspecto de contenidos: las traducciones de texto muy simples tales como cambiar meramente «Close» **por** la palabra «Fermer». Puesto que el equipo que realizaba las comprobaciones no hablaba de forma fluida el lenguaje en el que se pretendía trabajar, teníamos que dejar estos aspectos a traductores expertos del lenguaje.

El segundo aspecto de los catálogos del mensaje era la estructura, las reglas sintácticas que debía obedecer un catálogo de objetivos adecuadamente construido. A diferencia del contenido, en este caso sí que era posible para el equipo de comprobación el verificar los aspectos estructurales de los catálogos.

Como un ejemplo de lo que significa estructura, consideremos las etiquetas y los mnemónicos de un menú de aplicación. Un menú está constituido por etiquetas y sus mnemónicos (abreviaturas) asociados. Cada menú, independientemente de su contenido o su localización, debe obedecer las siguientes reglas listadas en la Guía de Estilo Motif

- Cada nemotécnico debe estar contenido en su etiqueta asociada;
- Cada nemotécnico debe ser único dentro del menú;
- Cada nemotécnico debe ser un carácter Único;
- Cada nemotécnico debe estar en ASCII.

Estas reglas son invariantes a través de las localizaciones, y pueden ser utilizadas para verificar que un menú está correctamente construido en la localización objetivo.

Existen varias posibilidades para realizar la prueba de errores de los mnemónicos del menú:

Dispositivo de prevención. Nosotros podemos escribir un programa para generar los mnemónicos automáticamente, dada una lista de etiquetas en cada menú. Este enfoque evitaría errores, pero el problema es que escoger un mnemónico adecuado es difícil, y el esfuerzo requerido para escribir el programa no estaría justificado con el beneficio obtenido.

Dispositivo de prevención. Podríamos escribir un programa que prevendría al localizador de elegir unos mnemónicos que no satisfagan el criterio. Este enfoque también evitaría errores, pero el beneficio obtenido sería mínimo: los mnemónicos incorrectos son lo suficiente-

mente fáciles de detectar y corregir después de que aparezcan.

Dispositivo de detección. Nosotros podríamos proporcionar un programa para verificar que las etiquetas del menú escogidas y los mnemónicos satisfacen los criterios anteriores. Nuestros localizadores podrían ejecutar los programas sobre los catálogos de mensaje traducidos antes de enviarnos a nosotros dichos catálogos. Este enfoque proporcionaría una realimentación rápida sobre los errores y será, probablemente, un paso a dar en el futuro.

Dispositivo de detección. Nosotros podríamos escribir un programa que verificase las etiquetas y mnemónicos del menú, y ejecutara el programa sobre catálogos de mensajes después de que nos los hayan devuelto los localizadores. Este enfoque es el camino que actualmente estamos tomando. No es tan eficiente como alguno de los métodos anteriormente mencionados y puede requerir que se establezca una comunicación fluida hacia delante y hacia atrás con los localizadores, pero los errores detectados son incluso fáciles de corregir en este punto.

Varios textos pequeños de poka-yoke se utilizaron como dispositivos poka-yoke para validar los aspectos estructurales de los menús. Un pequeño «script» *poka-yoke* leería la tabla, recuperaría los mnemónicos y etiquetas a partir del catálogo de mensajes, y compararía posteriormente las cadenas recuperadas con el criterio establecido descrito anteriormente.

Los «scripts» poka-yoke eran pequeños (aproximadamente 100 líneas), fáciles de escribir (algunos de los escritos estaban en cuanto al tiempo por debajo de una hora) y fáciles de ejecutar. Nosotros ejecutábamos nuestros «scripts» *poka-yoke* sobre 16 aplicaciones en la ubicación en inglés por defecto y en 11 ubicaciones extranjeras. Cada ubicación contenía 100 menús para un total de 1.200 menús. Los dispositivos *poka-yoke* encontraron 311 errores en menús y mnemónicos. Pocos de los problemas que nosotros descubrimos eran como muy llamativos, pero en total habrían supuesto una gran preocupación en la prueba y ejecución de nuestras aplicaciones localizadas.

El ejemplo descrito antes representa un dispositivo poka-yoke que ha sido integrado en la actividad de pruebas de ingeniería del software. La técnica poka-yoke puede aplicarse a los niveles de diseño, codificación y pruebas y proporciona un filtro efectivo de garantía de calidad.

8.9 EL ESTÁNDAR DE CALIDAD ISO 9001

Esta sección contiene varios objetivos, siendo el principal describir el cada vez más importante estándar internacional ISO 9001. El estándar, que ha sido adoptado por más de 130 países para su uso, se está convirtiendo en el medio principal con el que los clientes pueden juzgar la competencia de un desarrollador de software. Uno de los problemas con el estándar **ISO** 9001 está en que no es específico de la industria: está expresado en términos generales, y puede ser interpretado por los desarrolladores de diversos productos como cojinetes de bolas (rodamientos), secadores de pelo, automóviles, equipamientos deportivos y televisiones, así como por desarrolladores de software. Se han realizado muchos documentos que relacionan el estándar con la industria del software, pero no entran en una gran cantidad de detalles. El objetivo de esta sección es describir lo que significa el **ISO** 9001 en términos de elementos de calidad y técnicas de desarrollo.

Para la industria del software los estándares relevantes son:

- *ISO* 9001. Quality Systems-Model *for* Quality *Assurance* in Design, Development, Production, *Installation* and Servicing. Este es un estándar que describe el sistema de calidad utilizado para mantener el desarrollo de un producto que implique diseño.
- *ISO* 9000-3. Guidelines *for* Application *of ISO* 9001 to the Development, *Supply* and Maintainance *of Software*. Este es un documento específico que interpreta el **ISO** 9001 para el desarrollador de software.
- *ISO* 9004-2. Quality Management and Quality *System* Elements —*Part* 2—. Este documento proporciona las directrices para el servicio de facilidades del software como soporte de usuarios.

Los requisitos se agrupan bajo 20 títulos:

Responsabilidad de la gestión.
Inspección, medición y equipo de pruebas.
Sistema de calidad.
Inspección y estado de pruebas.
Revisión de contrato.
Acción correctiva.
Control de diseño.
Control de producto no aceptado.
Control de documento.
Tratamiento, almacenamiento, empaquetamiento y entrega.
Compras.
Producto proporcionado al comprador.
Registros de calidad.
Identificación y posibilidad de seguimiento del producto,
Auditorías internas de calidad.
Formación
Control del proceso
Servicios.
Inspección y estado de prueba.
Técnicas estadísticas.

Merece la pena ver un pequeño extracto de la **ISO** 9001. Este le dará al lector una idea del nivel con el que la ISO 9001 trata la garantía de calidad y el proceso de

desarrollo. El extracto elegido proviene de la sección 1.11:

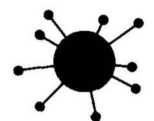
> **4.11. El equipo de Inspección, medición y pruebas.**
>
> **El suministrador debe controlar, calibrar y mantener la inspección, medir y probar el equipo, ya sea el dueño el suministrador, prestado o proporcionado por el comprador, para demostrar la conformidad del producto con los requisitos especificados. El equipo debe utilizarse de un modo que asegure que se conoce la incertidumbre de la medición y que es consistente con la capacidad de medición requerida.**

Lo primero a destacar es su generalidad; se puede aplicar al desarrollador de cualquier producto. Lo segundo a considerar es la dificultad en la interpretación del párrafo —es pretendido obviamente por los procesos estándar de ingeniería donde equipos tales como indicadores de calibración y potenciómetros son habituales—.

Una interpretación del párrafo anterior es que el distribuidor debe asegurar que cualquiera de las herramientas de software utilizadas para las pruebas tiene, por lo menos, la misma calidad que el software a desarrollar, y que cualquier prueba del equipo produce valores de medición, por ejemplo, los monitores del rendimiento, tienen una precisión aceptable cuando se compara con la precisión especificada para el rendimiento en la especificación de los requisitos.

8.10 EL PLAN DE SQA

El *plan de* SQA proporciona un mapa para institucionalizar la garantía de calidad del software. El plan, desarrollado por un grupo de SQA, sirve como plantilla para actividades de SQA instituidas para cada proyecto de software.



El Plan GCS

El IEEE [IEEE94] ha recomendado un estándar para los planes de SQA. Las secciones iniciales describen el propósito y el alcance del documento e indican aquellas actividades del proceso del software cubiertas por la garantía de calidad. Se listan todos los documentos señalados en el *plan de SQA* y se destacan todos los estándares aplicables. La sección de *Gestión* del plan describe la situación de la SQA dentro de la estructura organizativa; las tareas y las actividades de SQA y su emplazamiento a lo largo del proceso del software; así como los papeles y responsabilidades organizativas relativas a la calidad del producto.

La sección de *Documentación* describe (por referencia) cada uno de los productos de trabajo producidos como parte del proceso de software. Entre estos se incluyen:

- documentos del proyecto (por ejemplo: plan del proyecto),
- modelos (por ejemplo: DERs, jerarquías de clases),
- documentos técnicos (por ejemplo: especificaciones, planes de prueba),
- documentos de usuario (por ejemplo: archivos de ayuda).

Además, esta sección define el conjunto mínimo de productos de trabajo que se pueden aceptar para lograr alta calidad.

Los *estándares, prácticas* y *convenciones* muestran todos los estándares/prácticas que se aplican durante el proceso de software (por ejemplo: estándares de documentos, estándares de codificación y directrices de revisión). Además, se listan todos los proyectos, procesos y (en algunos casos) métricas de producto que se van a recoger como parte del trabajo de ingeniería del software.

La sección *Revisiones* y *Auditorías* del plan identifica las revisiones y auditorías que se van a llevar a cabo por el equipo de ingeniería del software, el grupo de SQA y el cliente. Proporciona una visión general del enfoque de cada revisión y auditoría.

La sección *Prueba* hace referencia al *Plan* y *Procedimiento de Pruebas del Software* (Capítulo 18). También define requisitos de mantenimiento de registros de pruebas. La *Información sobre problemas* y *acción correctiva* define procedimientos para informar, hacer seguimiento y resolver errores y defectos, e identifica las responsabilidades organizativas para estas actividades.

El resto del *plan de SQA* identifica las herramientas y métodos que soportan actividades y tareas de SQA; hace referencia a los procedimientos de gestión de configuración del software para controlar el cambio; define un enfoque de gestión de contratos; establece métodos para reunir, salvaguardar y mantener todos los registros; identifica la formación que se requiere para cumplir las necesidades del plan y define métodos para identificar, evaluar, supervisar y controlar riesgos.

RESUMEN

La garantía de calidad del software es una «actividad de protección» que se aplica a cada paso del proceso del software. La **SQA** comprende procedimientos para la aplicación efectiva de métodos y herramientas, revisiones técnicas formales, técnicas y estrategias de prueba, dispositivos ***poku-yoke,*** procedimientos de control de cambios, procedimientos de garantía de ajuste a los estándares y mecanismos de medida e información.

La **SQA** es complicada por la compleja naturaleza de la calidad del software —un atributo de los programas de computadora que se define como «concordancia con los requisitos definidos explícita e implícitamente» —. Cuando se considera de forma más general, la calidad del software engloba muchos factores de proceso y de producto diferentes con sus métricas asociadas.

Las revisiones del software son una de las actividades más importantes de la **SQA.** Las revisiones sirven como filtros durante todas las actividades de ingeniería del software, eliminando defectos mientras que no son relativamente costosos de encontrar y corregir. La revisión técnica formal es una revisión específica que se ha mostrado extremadamente efectiva en el descubrimiento de errores.

Para llevar a cabo adecuadamente una garantía de calidad del software, se deben recopilar, evaluar y distribuir todos los datos relacionados con el proceso de ingeniería del software. La **SQA** estadística ayuda a mejorar la calidad del producto y la del proceso de software. Los modelos de fiabilidad del software amplían las medidas, permitiendo extrapolar los datos recogidos sobre los defectos, a predicciones de tasas de fallo y de fiabilidad.

Resumiendo, recordemos las palabras de Dunn y Ullman [DUN82]: «La garantía de calidad del software es la guía de los preceptos de gestión y de las disciplinas de diseño de la garantía de calidad para el espacio tecnológico y la aplicación de la ingeniería del software.)) La capacidad para garantizar la calidad es la medida de madurez de la disciplina de ingeniería. Cuando se sigue de forma adecuada esa guía anteriormente mencionada, lo que se obtiene ***es*** la madurez de la ingeniería del software.

REFERENCIAS

[ALV64] von Alvin, W. H. (ed.), *Reliability Engineering,* Prentice-Hall, 1964.

[ANS87] ANSI/ASQC A3-1987, Quality Systems Terminology, 1987.

[ART92] Arthur, L. J., *Improving Software Quality: An Insider's Guide to TQM,* Wiley, 1992.

[ART97] Arthur, L. J., «Quantum Improvements in Software System Quality», *CACM,* vol. 40, n.º 6, Junio 1997, pp. 47-52.

[BOE81] Boehm, B., *Software Engineering Economics,* Prentice-Hall, 1981.

[CR075] Crosby, P., *Quality is Free,* McGraw-Hill, 1975.

[CR079] Crosby, P., *Quality is Free,* McGraw-Hill, 1979.

[DEM86] Deming, W. W., *Out of the Crisis,* MIT Press, 1986.

[DEM99] DeMarco, T., «Management Can Make Quality (Im)possible», presentación de Cutter Summit '99, Boston, MA, 26 de Abril 1999.

[DIJ76] Dijkstra, E., *A Discipline of Programming,* Prentice-Hall, 1976.

[DUN82] Dunn, R., y R. Ullman, *Quality Assurance for Computer Software,* McGraw-Hill, 1982.

[FRE90] Freedman, D. P., y G. M. Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews,* 3.ª ed., Dorset House, 1990.

[GIL93] Gilb, T., y D. Graham, *Software Inspections,* Addison-Wesley, 1993.

[GLA98] Glass, R., «Defining Quality Intuitively», *IEEE Software,* Mayo 1998, pp. 103-104 y 107.

[HOY98] Hoyle, D., *Iso 9000 Quality Systems Development Handbook: A Systems Engineering Approach,* Butterworth-Heinimann, 1998.

[IBM81] «Implementating Software Inspections», notas del curso, IBM Systems Sciences Institute, IBM Corporation, 1981.

[IEE90] *Software Quality Assurance: Model Procedures,* Institution of Electrical Engineers, 1990.

[IEE94] *Software Engineering Standards,* ed. de 1994, IEEE Computer Society, 1994

[JAN86] Jahanian, F., y A. K. Mok, «Safety Analysis of Timing Properties of Real Time Systems», *IEEE Trans. Software Engineering,* vol. SE-12, n.° 9, Septiembre 1986, pp. 890-904.

[JON86] Jones, T. C., *Programming Productivity,* McGraw-Hill, 1986.

[KAN95] Kan, S. H., *Metrics and Models in Software Quality Engineering,* Addison Wesley, 1995.

[KAP95] Kaplan, C., R. Clark y V. Tang, *Secrets of Software Quality: 40 Innovations from IBM,* McGraw-Hill, 1995.

[LEV86] Leveson, N.G., «Software Safety: Why, What, and How», *ACM Computing Surveys,* vol. 18, n.º 2, Junio 1986, pp. 125-163.

[LEV87] Leveson, N. G., y J.L. Stolzy, «Safety Analysis using Petri Nets», *IEEE Trans. Software Engineering, vol.* SE-13, n.° 3, Marzo 1987, **pp.** 386-397.

[LEV95] Leveson, N.G., *Safeware: System Safety and Computers,* Addison-Wesley, 1995.

[LIN79] Linger, R., H. Mills y B. Witt, *Structured Programming*, Addison-Wesley, 1979.

[LIT89] Littlewood, B., «Forecasting Software Reliability», en *Software Reliahility: Modelling and ldentification* (S. Bittanti, ed.), Springer-Verlag, 1989, pp. 141-209.

[MAN96] Manns, T. y M. Coleman, *Software Quality Assurance,* MacMillan Press, 1996.

[MUS87] Musa, J. D., A. Iannino y K. Okumoto, *Engineering and Managing Software with Reliahility Measures,* McGraw-Hill, 1987.

[PAU93] Paulk, M., et al., *Capability Maturiry Model for Software,* Software Engineering Institute, Camegie Mellon University, Pittsburgh, PA, 1993.

[POR95] Porter, A., H. Siv, C. A. Toman, y L. G. Votta, «An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development», *Proc. Third ACM SIGSOFT Symposium on the Foundations of Software Engineering,* Washington DC, Octubre 1996, ACM Press, pp. 92-103.

[ROB97] Robinson, H., «Using Poka-Yoke Techniques for Early Error Detection», *Proc. Sixth International Conference on Software Testing Analysis and Review* (STAR'97), 1997, pp. 119-142.

[ROO90] Rook, J., *Software Reliahility Handhook,* Elsevier, 1990.

[SCH98] Schulmeyer, G. C., y J.I. McManus (eds.), *Handhook of Software Quality Assurance,* Prentice-Hall, 3.ª ed., 1998.

[SCH94] Schmauch, C.H., *ISO 9000 for Software Developers,* ASQC Quality Press, Milwaukee, WI, 1994.

[SCH97] Schoonmaker, S.J., *ISO 9000 for Engineers and Designers,* McGraw Hill, 1997.

[SHI86] Shigeo Shingo, *Zero Quality Control: Source Inspection and the Poka-Yoke System,* Productivity Press, 1986.

[SOM96] Somerville, I., *Software Engineering,* 5.ª ed., Addison-Wesley, 1996.

[TIN96] Tingey, M., *Comparing ISO 9000,* Malcolm Baldridge y al SEJ CMM para Software, Prentice-Hall, 1996.

[TRI97] Tricker, R., *ISO 9000 for Small Bussinesses,* Butterworth-Heinemann, 1997.

[VES81] Veseley, W.E., et al., *Fault Tree Handhook, U.S* Nuclear Regulatory Commision, NUREG-0492, Enero 1981.

[WI494] Wilson, L. A., *8 stp to Succesful ISO 9000,* Cambridge Interactive Publications, 1996.

PROBLEMAS Y PUNTOS A CONSIDERAR

8.1. Al principio del capítulo se señaló que «el control de variación está en el centro del control de calidad». Como todos los programas que se crean son diferentes unos de otros, ¿cuáles son las variaciones que se buscan y cómo se controlan?

8.2. ¿Es posible evaluar la calidad del software si el cliente no se pone de acuerdo sobre lo que se supone que ha de hacer?

8.3. La calidad y la fiabilidad son conceptos relacionados, pero son fundamentalmente diferentes en varias formas. Discútalas.

8.4. ¿Puede un programa ser correcto y aún así no ser fiable? Explique por qué.

8.5. ¿Puede un programa ser correcto y aun así no exhibir una buena calidad? Explique por qué.

8.6. ¿Por qué a menudo existen fricciones entre un grupo de ingeniería del software y un grupo independiente de garantía de calidad del software? ¿Es esto provechoso?

8.7. Si se le da la responsabilidad de mejorar la calidad del software en su organización. ¿Qué es lo primero que haría? ¿Qué sería lo siguiente?

8.8. Además de los errores, ¿hay otras características claras del software que impliquen calidad? ¿Cuáles son y cómo se pueden medir directamente?

8.9. Una revisión técnica formal sólo es efectiva si todo el mundo se la prepara por adelantado. ¿Cómo descubriría que uno de los participantes no se la ha preparado? ¿Qué haría si fuera el jefe de revisión?

8.10. Algunas personas piensan que una RTF debe evaluar el estilo de programación al igual que la corrección. ¿Es una buena idea? ¿Por qué?

8.11. Revise la Tabla 8.1 y seleccione las cuatro causas vitales de errores serios y moderados. Sugiera acciones correctoras basándose en la información presentada en otros capítulos.

8.12. Una organización utiliza un proceso de ingeniería del software de cinco pasos en el cual los errores se encuentran de acuerdo a la siguiente distribución de porcentajes:

Paso	Porcentaje de errores encontrados
1	20 %
2	15 %
3	15 %
4	40 %
5	10 %

Mediante la información de la Tabla 8.1 y la distribución de porcentajes anterior, calcule el índice de defectos global para dicha organización. Suponga que PS = 100.000.

8.13. Investigue en los libros sobre fiabilidad del software y escriba un artículo que explique un modelo de fiabilidad del software. Asegúrese de incluir un ejemplo.

8.14. El concepto de TMEF del software sigue abierto a debate. ¿Puede pensar en algunas razones para ello?

8.15. Considere dos sistemas de seguridad crítica que estén controlados por una computadora. Liste al menos tres peligros para cada uno de ellos que se puedan relacionar directamente con los fallos del software.

8.16. Utilizando recursos web y de impresión, desarrolle un tutorial de 20 minutos sobre *poka-yoke* y expóngaselo a su clase.

8.17. Sugiera unos cuantos dispositivos *poka-yoke* que puedan ser usados para detectar y/o prevenir errores que son encontrados habitualmente antes de «enviar» un mensaje por e-mail.

8.18. Adquiera una copia de ISO 9001 e ISO 9000-3. Prepare una presentación que trate tres requisitos ISO 9001 y cómo se aplican en el contexto del software.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Los libros de Moriguchi (*Software Excellence: A Total Quality Management Guide,* Productivity Press, 1997), Horch (*Practica1 Guide to Software Quality Management,* Artech Publishing, 1996) son unas excelentes presentaciones a nivel de gestión sobre los beneficios de los programas formales de garantía de calidad para el software de computadora. Los libros de Deming [DEM86] y Crosby [CRO79], aunque no se centran en el software, ambos libros son de lectura obligada para los gestores senior con responsabilidades en el desarrollo de software. Gluckman y Roome (*Every day Heroes of the Quulity Movement,* Dorset House, 1993) humaniza los aspectos de calidad contando la historia de los participantes en el proceso de calidad. Kan (*Metrics and Models in Software Quality Engineering,* Addison-Wesley, 1995) presenta una visión cuantitativa de la calidad del software. Manns (*Software Quality Assurance,* Macmillan, 1996) es una excelente introducción a la garantía de calidad del software.

Tingley (*Compering ISO 9000, Malcom Baldridge, and the SEI CMM for Software,* Prentice-Hall, 1996) proporciona una guía útil para las organizaciones que se esfuerzan en mejorar sus procesos de gestión de calidad. Oskarsson (*An ISO 9000 Approach to Building Quality Software,* Prentice-Hall, 1995) estudia cómo se aplica el estándar ISO al software.

Durante los últimos años se han escrito docenas de libros sobre aspectos de garantía de calidad. La lista siguiente es una pequeña muestra de fuentes útiles:

Clapp, J. A., et al., *Software Quality Control, Error Analysis and Testing,* Noyes Data Corp.. Park Ridge, NJ, 1995.

Dunn, R. H., y R.S. Ullman, *TQM for Computer Software,* McGrawHill, 1994.

Fenton, N., R. Whitty y Y. Iizuka, *Software Quality Assurance and Measurement: Worldwide Industrial Applications,* Chapman & Hall, 1994.

Ferdinand, A. E., *Systems, Software, and Quality Engineering,* Van Nostrand Reinhold, 1993.

Ginac, F. P., *Customer Oriented Software Quality Assurance,* Prentice-Hall, 1998.

Ince, D., *ISO 9001 and Software Quality Assurance,* McGraw-Hill, 1994.

Ince, D., *An lntroduction to Software Quality Assurance and Implementation,* McGraw-Hill, 1994.

Jarvis, A., y V. Crandall, *Inroads to Software Quality: «How To» Guide and Toolkit,* Prentice-Hall, 1997.

Sanders, J., *Software Quality: A Framework for Success in Sofiware Developrnent,* Addison-Wesley, 1994.

Sumner, F. H. *Software Quality Assurance,* MacMillan, 1993.

Wallmuller, E., *Software Quality Assurance: A Practical Approach,* Prentice-Hall, 1995.

Weinberg, G. M., *Quality Software Managenzent,* 4 vols., Dorset House, 1992, 1993, 1994 y 1996.

Wilson, R. C., *Software Rx: Secrets of Engineering Quality Sofiware,* Prentice-Hall, 1997.

Una antología editada por Wheeler, Bryczynsky y Meeson (*Software Inspection: lndustry Best Practice,* IEEE Computer Society Press, 1996) presenta información útil sobre esta importante actividad de GCS. Friedman y Voas (*Software Assessment*, Wiley, 1995) estudian los soportes y métodos prácticos para asegurar la fiabilidad y la seguridad de programas de computadora.

Musa (*Software Reliability Engineering: More Reliable Software, Faster Development and Testing,* McGraw-Hill, 1998) ha escrito una guía práctica para aplicar a las técnicas de fiabilidad del software. Han sido editadas antologías de artículos importantes sobre la fiabilidad del software por Kapur y otros (*Contrihutions to Hardware and Software Reability Modelling,* World Scientific Publishing Co., 1999), Gritzails (*Reliability, Quality and Safety of Software-Intensive Systems,* Kluwer Academic Publishing, 1997), y Lyu (*Handbook of Software Reliahility Engineering,* McGraw-Hill, 1996). Storey (*Safety-Critica1 Computer Systems,* Addison-Wesley, 1996) y Leveson [LEV95] continúan siendo los estudios más completos sobre la seguridad del software publicados hasta la fecha.

Además de [SHI86], la técnicapoka-yoke para el software de prueba de errores es estudiada por Shingo (*The Shingo Production Management System: Improving Product Quality by Preventing Defects,* Productivity Press, 1989) y por Shimbun (*Poka-Yoke: Improving Product Quulity by Preventing Defects,* Productivity Press, 1989).

En Intemet están disponibles una gran variedad de fuentes de información sobre la garantía de calidad del software, fiabilidad del software y otros temas relacionados. Se puede encontrar una lista actualizada con referencias a sitios (páginas) web que son relevantes para la calidad del software en **http://www.pressman5.com.**

CAPÍTULO

9 GESTIÓN DE LA CONFIGURACIÓN DEL SOFTWARE (GCS/SCM)*

CUANDO se construye software de computadora, los cambios son inevitables. Además, los cambios aumentan el grado de confusión entre los ingenieros del software que están trabajando en el proyecto. La confusión surge cuando no se han analizado los cambios antes de realizarlos, no se han registrado antes de implementarlos, no se les han comunicado a aquellas personas que necesitan saberlo o no se han controlado de manera que mejoren la calidad y reduzcan los errores. Babich [BAB86] se refiere a este asunto cuando dice:

> El arte de coordinar el desarrollo de software para minimizar...**la** confusión, se denomina gestión de configuración. La gestión de configuración es el arte de identificar, organizar y controlar **las** modificaciones que sufre el software que construye un equipo de programación. La meta es maximizar **la** productividad minimizando los errores.

La gestión de configuración del software (GCS) es una actividad de autoprotección que ***se*** aplica durante el proceso del software. Como el cambio se puede producir en cualquier momento, las actividades de GCS sirven para (1) identificar el cambio, (2) controlar el cambio, ***(3)*** garantizar que el cambio se implementa adecuadamente y (4) informar del cambio a todos aquellos que puedan estar interesados.

Es importante distinguir claramente entre el mantenimiento del software **y** la gestión de configuración del software. El mantenimiento es un conjunto de actividades de ingeniería del software que se producen después de que el software se haya entregado al cliente y esté en funcionamiento. La gestión de configuración del software es un conjunto de actividades de seguimiento y control que comienzan cuando se inicia el proyecto de ingeniería del software y termina sólo cuando el software queda fuera de la circulación.

VISTAZO RÁPIDO

¿Qué *es?* Cuando construimos software de computadora, surgen cambios. Debido a esto, necesitamos controlarlos eficazmente. La gestión de la configuración del software (GCS) es un conjunto de actividades diseñadas para controlar el cambio identificando los productos del trabajo que probablemente cambien, estableciendo relaciones entre ellos, definiendo mecanismos para gestionar distintas versiones de estos productos, controlando los cambios realizados, y auditando e informando de los cambios realizados.

¿Quién lo hace? Todos aquellos que estén involucrados en el proceso de ingeniería de software están relacionados con la GCS hasta cierto punto, pero las posiciones de mantenimiento especializadas son creadas a veces para la gestión del proceso de GCS.

¿Por qué es importante? Si no controlamos el cambio, él nos controlará a nosotros. Y esto nunca es bueno. Es muy fácil para un flujo de cambios incontrolados llevar al caos a un proyecto de software correcto. Por esta razón, la GCS es una parte esencial de una buena gestión del proyecto y una práctica formal de la ingeniería del software.

¿Cuáles son los pasos? Puesto que muchos productos de trabajo se obtienen cuando se construye el software, cada uno debe estar identificado unívocamente. Una vez que esto se ha logrado, se pueden establecer los mecanismos para el control del cambio y de las versiones. Para garantizar que se mantiene la calidad mientras se realizan los cambios, se audita el proceso. y para asegurar que aquellos que necesitan conocer los cambios son informados, se realizan los informes.

¿Cuál es el producto obtenido? El Plan de Gestión de la Configuración del Software define la estrategia del proyecto para la GCS. Además, cuando se realiza la GCS formal, el proceso de control del cambio provoca peticiones de cambio del software e informes de órdenes de cambios de ingeniería.

¿Cómo puedo estar *seguro* de que lo he hecho correctamente? Cuando cualquier producto de trabajo puede ser estimado para ser supervisado y controlado: cuando cualquier cambio pueda ser seguido y analizado; cuando cualquiera que necesite saber algo sobre algún cambio ha sido informado, lo habremos realizado correctamente.

* En inglés, Software Configuration Management.

9.1 GESTIÓN DE LA CONFIGURACIÓN DEL SOFTWARE

El resultado del proceso de ingeniería del software es una información que se puede dividir en tres amplias categorías: (1) programas de computadora (tanto en forma de código fuente como ejecutable), (2) documentos que describen los programas de computadora (tanto técnicos como de usuario) y *(3)* datos (contenidos en el programa o externos a él). Los elementos que componen toda la información producida como parte del proceso de ingeniería del software se denominan colectivamente configuración del software.

A medida que progresa el proceso del software, el número de *elementos de configuración del software (ECSs)* crece rápidamente. Una especificación del sistema produce un plan del proyecto del software y una especificación de requisitos del software (así como otros documentos relativos al hardware). A su vez, éstos producen otros documentos para crear una jerarquía de información. Si simplemente cada ECS produjera otros ECSs, no habría prácticamente confusión. Desgraciadamente, en el proceso entra en juego otra variable —el cambio—. El cambio se puede producir en cualquier momento y por cualquier razón. De hecho, la Primera Ley de la Ingeniería de Sistemas [BER80] establece: Sin importar en qué momento del ciclo de vida del sistema nos encontremos, el sistema cambiará y el deseo de cambiarlo persistirá a lo largo de todo el ciclo de vida.



No hay nada permanente excepto el cambio
Heráclito, 500 AdC.

¿Cuál es el origen de estos cambios? La respuesta a esta pregunta es tan variada como los cambios mismos. Sin embargo, hay cuatro fuentes fundamentales de cambios:

- nuevos negocios o condiciones comerciales que dictan los cambios en los requisitos del producto o en las normas comerciales;
- nuevas necesidades del cliente que demandan la modificación de los datos producidos por sistemas de información, funcionalidades entregadas por productos o servicios entregados por un sistema basado en computadora;
- reorganización o crecimiento/reducción del negocio que provoca cambios en las prioridades del proyecto o en la estructura del equipo de ingeniería del software;
- restricciones presupuestarias o de planificación que provocan una redefinición del sistema o producto.

La gestión de configuración del software (GCS) es un conjunto de actividades desarrolladas para gestionar los cambios a lo largo del ciclo de vida del software de computadora.



La mayoría de los cambios son justificados. No lamente los cambios. Mejor dicho, esté seguro que tiene los mecanismos preparados para realizarlos.

9.1.1. Líneas base

Una línea base es un concepto de gestión de configuración del software que nos ayuda a controlar los cambios sin impedir seriamente los cambios justificados. **La** IEEE (Estándar IEEE 610.12-1990) define una línea base como:

> Una especificación o producto que se ha revisado formalmente y sobre los que se **ha** llegado a un acuerdo, y que de ahí en adelante sirve como base para un desarrollo posterior y que puede cambiarse solamente a través de procedimientos formales de control de cambios.

Una forma de describir la línea base es mediante una analogía:

> Considere las puertas de la cocina en un gran restaurante. Para evitar colisiones, una puerta esta marcada como **SALIDA** y la otra como ENTRADA. Las puertas tienen topes que hacen que sólo se puedan abrir en la dirección apropiada.
>
> Si un camarero recoge un pedido en la cocina, lo coloca en una bandeja luego se da cuenta de que ha cogido un plato equivocado, puede cambiar el plato correcto rápida e informalmente antes de salir de la cocina.
>
> Sin embargo, si abandona la cocina, le da el plato **al** cliente y luego se le informa de su error, debe seguir el siguiente procedimiento: (1) mirar en **la** orden de pedido si ha habido algún error; *(2)* disculparse insistentemente; (3) volver a la cocina por la puerta de ENTRADA; (4) explicar el problema, etc.



Un producto de trabajo de la ingeniería del software se convierte en una línea base, solamente después de haber sido revisado y aprobado.

Una línea base es análoga a la cocina de un restaurante. Antes de que un elemento de configuración de software se convierta en una línea base, el cambio se puede llevar a cabo rápida e informalmente. Sin embargo, una vez que se establece una línea base, pasamos, de forma figurada, por una puerta de un solo sentido. Se pueden llevar a cabo los cambios, pero se debe aplicar un procedimiento formal para evaluar y verificar cada cambio.

En el contexto de la ingeniería del software, definimos una *línea base* como un punto de referencia en el desarrollo del software que queda marcado por el envío de uno o más elementos de configuración del software y la aprobación del ECS obtenido mediante una revisión técnica formal (Capítulo 8). Por ejemplo, los ele-

mentos de una *Especificación de Diseño* se documentan y se revisan. Se encuentran errores y se corrigen. Cuando todas las partes de la especificación se han revisado, corregido y aprobado, la *Especificación de Diseño* se convierte en una línea base. Sólo se pueden realizar cambios futuros en la arquitectura del software (documentado en la *Especificación de Diseño*) tras haber sido evaluados y aprobados. Aunque se pueden definir las líneas base con cualquier nivel de detalle, las líneas base más comunes son las que se muestran en la Figura 9.1.



Asegúrese que la base de datos del proyecta se montiene sobre un entorno ***controlado, centralizado.***

La progresión de acontecimientos que conducen a un línea base está también ilustrada en la Figura 9.1. Las tareas de la ingeniería del software producen uno o más ECSs. Una vez que un ECS se ha revisado y aprobado, se coloca en una *base de datos del proyecto* (también denominada *biblioteca del proyecto o depósito de software).* Cuando un miembro del equipo de ingeniería del software quiere hacer modificaciones en un ECS de línea base, **se** copia de la base de datos del proyecto a un área de trabajo privada del ingeniero. Sin embargo, este ECS extraído puede modificarse sólo si se siguen los controles GCS (tratados más adelante en este capítulo). Las flechas punteadas de la Figura 9.1 muestran el camino de modificación de una línea base ECS.

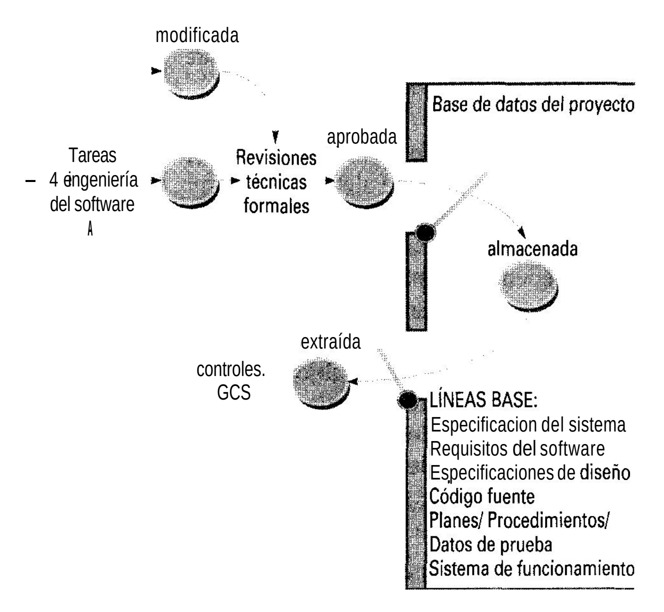


FIGURA 9.1. ECS de línea base **y** base de datos del proyecto.

9.1.2. Elementos de configuración del software

Ya hemos definido un elemento de configuración del software como la información creada como parte del proceso de ingeniería del software. Llevado al extremo, se puede considerar un ECS como una sección individual de una gran especificación o cada caso de prueba de un gran conjunto de pruebas. De forma más realista, un ECS es un documento, un conjunto completo de casos de prueba o un componente de un programa dado (p. ej., una función de C++ o un paquete de ADA).

En realidad, los ECSs se organizan como *objetos de configuración* que han de ser catalogados en la base de datos del proyecto con un nombre único. Un objeto de configuración tiene un nombre y unos atributos y está «conectado» a otros objetos mediante relaciones. **De** acuerdo con la Figura 9.2, los objetos de configuración, **Especificación de Diseño, modelo de datos, componente N, código fuente** y **Especificación de Prueba,** están definidos por separado. Sin embargo, cada objeto está relacionado con otros como muestran las flechas. Una flecha curvada representa una *relación de composición.* Es decir, **modelo de datos y componente N** son parte del objeto Especificación de Diseño. Una flecha recta con dos puntas representa una interrelación. Si se lleva a cabo un cambio sobre el objeto **código fuente,** las interrelaciones permiten al ingeniero de software determinar qué otros objetos (y ECSs) pueden verse afectados[1].

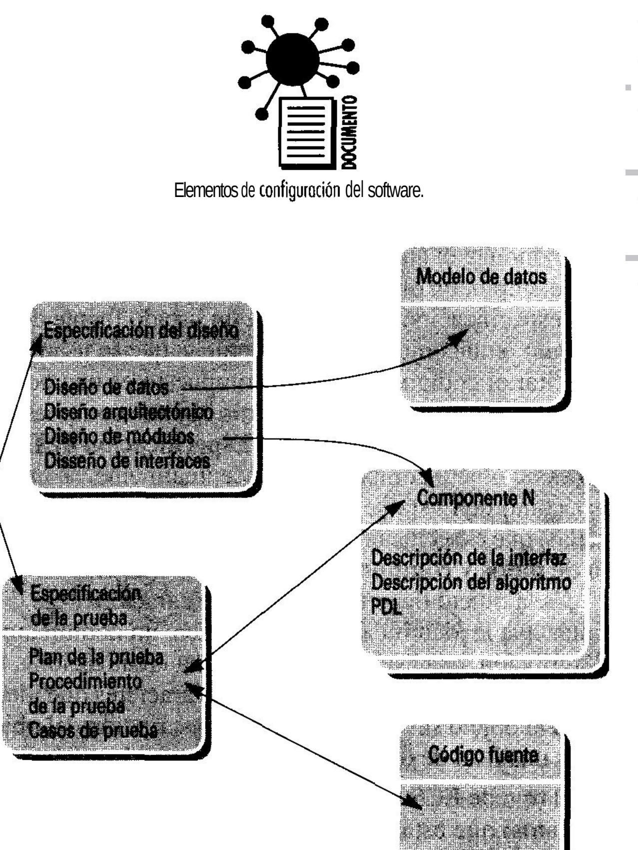
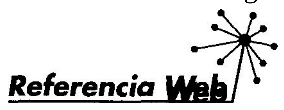


FIGURA 9.2. Objetos de configuración.

[1] Estas relaciones se tratan en la Sección 9.2.1 y la estructura de la base de datos del proyecto se trata con detalle en el Capítulo 31.

9.2 EL PROCESO DE GCS

La gestión de configuración del software es un elemento importante de garantía de calidad del software. Su responsabilidad principal es el control de cambios. Sin embargo, la GCS también es responsable de la identificación de ECSs individuales y de las distintas versiones del software, de las auditorías de la configuración del software para asegurar que se desarrollan adecuadamente y de la generación de informes sobre todos los cambios realizados en la configuración.



Las páginas amarillas de gestión de configuración contienen el listado más complejo de los recursos de GCS en la web. Para más información, consultar: **www.cs.colorado.edu/users/andre/ tonfiguration-managementhtml**

Cualquier estudio de la GCS plantea una serie de preguntas complejas:

- ¿Cómo identifica y gestiona una organización las diferentes versiones existentes de un programa (y su documentación) de forma que se puedan introducir cambios eficientemente?
- ¿Cómo controla la organización los cambios antes y después de que el software sea distribuido al cliente?
- ¿Quién tiene la responsabilidad de aprobar y de asignar prioridades a los cambios?
- ¿Cómo podemos garantizar que los cambios se **han** llevado a cabo adecuadamente?
- ¿Qué mecanismo se usa para avisar a otros de los cambios realizados?

Estas cuestiones nos llevan a la definición de cinco tareas de GCS: *Identificación, control de versiones, control de cambios, auditorías de configuración y generación de informes.*

9.3 IDENTIFICACIÓN DE OBJETOS EN LA CONFIGURACIÓN DEL SOFTWARE

Para controlar y gestionar los elementos de configuración, se debe identificar cada uno de forma Única y luego organizarlos mediante un enfoque orientado a objetos. Se pueden identificar dos tipos de objetos [CHO89]: *objetos básicos y objetos compuestos*[2]. Un objeto básico es una «unidad de texto» creado por un ingeniero de software durante el análisis, diseño, codificación o pruebas. Por ejemplo, un objeto básico podría ser una sección de una especificación de requisitos, un listado fuente de un módulo o un conjunto de casos prueba que se usan para ejercitar el código. Un objeto compuesto es una colección de objetos básicos y de otros objetos compuestos. De acuerdo con la Figura 9.2, **la Especificación de Diseno** es un objeto compuesto. Conceptualmente, se puede ver como una lista de referencias con nombre (identificadas) que especifican objetos básicos, tales como **modelo de datos y componente N.**

Cada objeto tiene un conjunto de características distintas que le identifican de forma Única: un nombre, una descripción, una lista de recursos y una «realización». El nombre del objeto es una cadena de caracteres que identifica al objeto sin ambigüedad. La descripción del objeto es una lista de elementos de datos que identifican:

- el tipo de ECS (por ejemplo: documento, programa, datos) que está representado por el objeto;
- un identificador del proyecto;
- la información de la versión y/o el cambio;



Las relaciones establecidos entre los objetos de configuración permiten al ingeniero del software evaluar el impacto del cambio.

Los recursos son «entidades que proporciona, procesa, referencia o son, de alguna otra forma, requeridas por el objeto». [CHO89], por ejemplo, los tipos de datos, las funciones específicas e incluso los nombres de las variables pueden ser considerados recursos de objetos. La realización es una referencia a la «unidad de texto» para un objeto básico y nulo para un objeto compuesto.

La identificación del objeto de configuración también debe considerar las relaciones existentes entre los objetos identificados. Un objeto puede estar identificado como **<parte-de>** un objeto compuesto. La relación **<parte-de>** define una jerarquía de objetos. Por ejemplo, utilizando esta sencilla notación

Diagrama E-R 1.4 <parte-de> modelo de datos;
Modelo de datos <parte-de> especificación de diseño;

creamos una jerarquía de ECSs.

No es realista asumir que la Única relación entre los objetos de la jerarquía de objetos se establece mediante largos caminos del árbol jerárquico. En muchos casos, los objetos están interrelacionados entre ramas de la

[2] Como mecanismos para representar una versión completa de la configuración del software se ha propuesto el concepto de objeto agregado [GUS89]

jerarquía de objetos. Por ejemplo, el modelo de datos está interrelacionado con los diagramas de flujo de datos (suponiendo que se usa el análisis estructurado) y también está interrelacionado con un conjunto de casos de prueba para una clase particular de equivalencia. Las relaciones a través de la estructura se pueden representar de la siguiente forma:

Modelo de datos <interrelacionado> modelo de flujo de datos;
Modelo de datos <interrelacionado> caso de prueba de la clase m;

Referencia cruzada

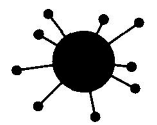
Los modelos de datos y los diagramas de flujo de datos se tratan en el Capítulo 12.

En el primer caso, la interrelación es entre objetos compuestos, mientras que en el segundo caso, la relación es entre un objeto compuesto **(modelo de datos)** y un objeto básico **(caso de prueba de la clase m).**

Las interrelaciones entre los objetos de configuración se pueden representar con un *lenguaje de interconexión de módulos (LIM)* [NAR87]. Un LIM describe las interdependencias entre los objetos de configuración y permite construir automáticamente cualquier versión de un sistema.

El esquema de identificación de los objetos de software debe tener en cuenta que los objetos evolucionan a lo largo del proceso de ingeniería del software. Antes de que un objeto se convierta en una línea base, puede cambiar varias veces, e incluso cuando ya es una línea base, los cambios se presentan con bastante frecuencia. **Se** puede crear un *grafo de evolución* [GUS89] para cualquier objeto. El grafo de evolución describe la historia de los cambios de un objeto y aparece ilustrado en la Figura 9.3. El objeto de configuración 1.0 pasa la revisión y se convierte en el objeto 1.1. Algunas correcciones y cambios mínimos producen las versiones 1.1.1 y 1.1.2, que van seguidas de una actualización importante, resultando el objeto 1.2. La evolución de objeto 1.0 sigue hacia 1.3 **y** 1.4, pero, al mismo tiempo, una gran modificación del objeto produce un nuevo camino de evolución, la versión 2.0. **A** estas dos versiones se les sigue dando soporte.

Se pueden realizar cambios en cualquier versión, aunque no necesariamente en todas. ¿Cómo puede el desarrollador establecer las referencias de todos los componentes, documentos, casos de prueba de la versión 1.4.? ¿Cómo puede saber el departamento comercial los nombres de los clientes que en la actualidad tienen la versión 2.1.? ¿Cómo podemos estar seguros de que los cambios en el código fuente de la versión 2.1 han sido reflejados adecuadamente en la correspondiente documentación de diseño? Un elemento clave para responder a todas estas preguntas es la identificación.



Herramientas CASE-GCS

Se han desarrollado varias herramientas automáticas para ayudar en la tarea de identificación (y otras de GCS). En algunos casos, se diseña la herramienta para mantener copias completas de las versiones más recientes.

9.4 CONTROL DE VERSIONES

El control de versiones combina procedimientos y herramientas para gestionar las versiones de los objetos de configuración creados durante el proceso del software. Clemm [CLE89] describe el control de versiones en el contexto de la GCS:



El esquema que Vd. estableció para los ECS debería incorporar el número de versión.

> **La** gestión de configuración permite a un usuario especificar configuraciones alternativas del sistema de software mediante la selección de las versiones adecuadas. Esto se puede gestionar asociando atributos a cada versión del software y permitiendo luego especificar [y construir] una configuración describiendo el conjunto de atributos deseado.

Los «atributos» que se mencionan pueden ser tan sencillos como un número específico de versión asociado a cada objeto o tan complejos como una cadena de variables lógicas (indicadores) que especifiquen tipos de cambios funcionales aplicados al sistema [LIE89].

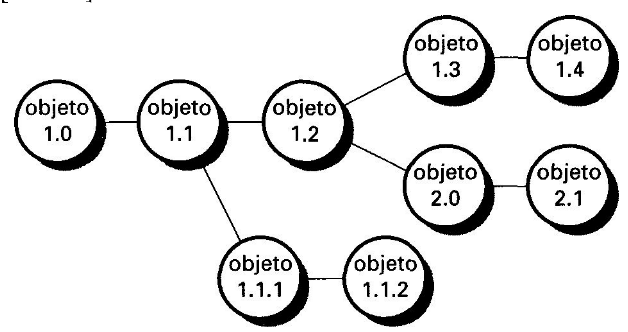


FIGURA 9.3. Grafo de evolución.

Una representación de las diferentes versiones de un sistema es el grafo de evolución mostrado en la Figura 9.3. Cada nodo del grafo es un objeto compuesto, es decir, una versión completa del software. Cada versión del software es una colección de ECSs (código fuente, documentos, datos) y cada versión puede estar com-

puesta de diferentes variantes. Para ilustrar este concepto, consideremos una versión de un sencillo programa que está formado por los componentes 1, 2, 3, 4 y 5[3]. El componente 4 sólo se usa cuando el software se implementa para monitores de color. El componente 5 se implementa cuando se dispone de monitor monocromo. Por tanto, se pueden definir dos variantes de la versión: (1) componentes 1, 2, 3 y 4; (2) componentes 1, 2, 3 y 5.

Para construir la ***variante*** adecuada de una determinada versión de un programa, a cada componente se le asigna una «tupla de atributos» —una lista de características que definen si se ha de utilizar el componente cuando se va a construir una determinada versión del software—. **A** cada variante se le asigna uno o más atributos. Por ejemplo, se podría usar un atributo color para definir qué componentes se deben incluir para soporte de monitores en color.

Cita:
Cualquier cambio, incluso un cambio para mejor, está siempre acompañado por desventajas y disconformidades.
Arnold Bennett

Otra forma de establecer los conceptos de la relación entre componentes, variantes y versiones (revisiones) es representarlas como un *fondo* de *objetos* [REI89]. De acuerdo con la Figura 9.4, la relación entre los objetos de configuración y los componentes, las variantes y las versiones se pueden representar como un espacio tridimensional. Un componente consta de una colección de objetos del mismo nivel de revisión. Una variante es una colección diferente de objetos del mismo nivel de revisión y, por tanto, coexiste en paralelo con otras variantes. Una nueva versión se define cuando se realizan cambios significativos en uno o más objetos.

En la pasada década se propusieron diferentes enfoques automatizados para el control de versiones. La principal diferencia entre los distintos enfoques está en la sofisticación de los atributos que se usan para construir versiones y variantes específicas de un sistema y en la mecánica del proceso de construcción.

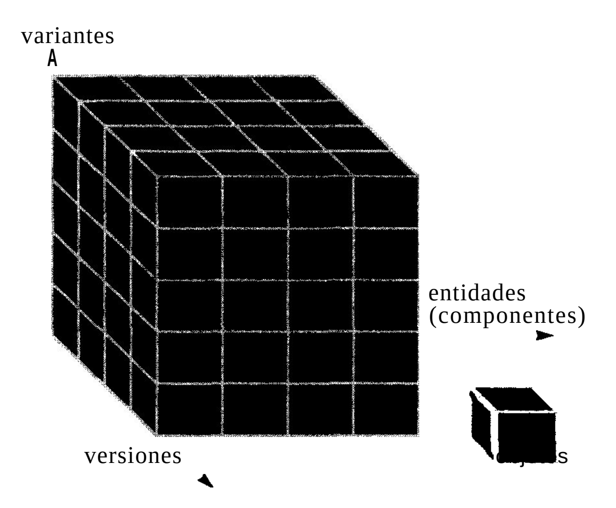


FIGURA 9.4. Representación en fondo de objetos de los componentes, variantes y versiones [REI89].

9.5 CONTROL DE CAMBIOS

La realidad del *control* de *cambio* en un contexto moderno de ingeniería de software ha sido bien resumida por James Bach [BAC98] :

> El control de cambio es vital. Pero las fuerzas que lo hacen necesario tambien lo hacen molesto. Nos preocupamos por el cambio porque una diminuta perturbacion en el código puede crear un gran fallo en el producto. Pero tambien puede reparar un gran fallo o habilitar excelentes capacidades nuevas. Nos preocupamos por el cambio porque un desarrollador pícaro puede hacer fracasar el proyecto; sin embargo **las** brillantes ideas nacidas en la mente de estos pícaros, y un pesado proceso de control de cambio pueden disuadirle de hacer un trabajo creativo.

Bach reconoce que nos enfrentamos a una situacion a equilibrar. Mucho control de cambio y crearemos problemas. Poco, y crearemos otros problemas.

En un gran proyecto de ingeniería de software, el cambio incontrolado lleva rápidamente al caos. Para estos proyectos, el control de cambios combina los procedimientos humanos y las herramientas automáticas para proporcionar un mecanismo para el control del cambio. El proceso de control de cambios está ilustrado esquemáticamente en la Figura 9.5. **Se** hace una *petición* de *cambio*[4] y se evalúa para calcular el esfuerzo técnico, los posibles efectos secundarios, el impacto global sobre otras funciones del sistema y sobre otros objetos de la configuración. Los resultados de la evaluación se presentan

Cita:
El arte del progreso está en mantener el orden en medio del cambio y mantener el cambio en medio del orden.
Alfred North Whitehead

[3] En este contexto, el término «componente» se refiere a todos los objetos compuestos y objetos básicos di: un ESC de línea base. Por ejemplo, un componente de «entrada» puede estar constituido por seis componentes de software distintos, cada uno responsable de una subfunción de entrada.

[4] Aunque muchas de las peticiones de cambio se reciben durante la fase de mantenimiento, en este estudio tomamos un punto de vista más amplio. Una petición de cambio puede aparecer en cualquier momento durante el proceso del software.

como un informe de cambios a la *autoridad de control de cambios (ACC)* —una persona o grupo que toma la decisión final del estado y la prioridad del cambio—. Para cada cambio aprobado se genera una *orden de cambio de ingeniería (OCI)*. La OCI describe el cambio a realizar, las restricciones que se deben respetar y los criterios de revisión y de auditoría. El objeto a cambiar es «dado de baja» de la base de datos del proyecto; se realiza el cambio y se aplican las adecuadas actividades de **SQA.** Luego, el objeto es «dado de alta» en la base de datos y se usan los mecanismos de control de versiones apropiados (Sección **9.4)** para crear la siguiente versión del software.

Se reconoce la necesidad del cambio
↓
El usuario suscribe la petición de cambio
↓
El desarrollador la evalua
↓
Se genera un informe de cambios
↓
La autoridad de control de cambios decide

La petición queda pendiente de actuación, se genera la OCI → Petición de cambio denegada → Información del usuario

La petición queda pendiente de actuación, se genera la OCI
↓
Asignación personalizada a los objetos de configuración
↓
"Dar de baja" objetos (elementos) de configuración
↓
Realización del cambio
↓
Revisión de cambio (auditoría)
↓
Los elementos de configuración que han cambiado son "dados de alta"
↓
Establecimiento de una línea base para la prueba
↓
Realización de actividades de garantía de calidad y de prueba
↓
"Promoción" de los cambios para ser incluidos en la siguiente versión (revisión)
↓
Reconstrucción de la versión adecuada del software
↓
Revisión (auditoría) de los cambios en todos los elementos de configuración
↓
Cambios incluidos en la nueva versión
↓
Distribución de la nueva versión

FIGURA 9.5. El proceso de control de cambios.

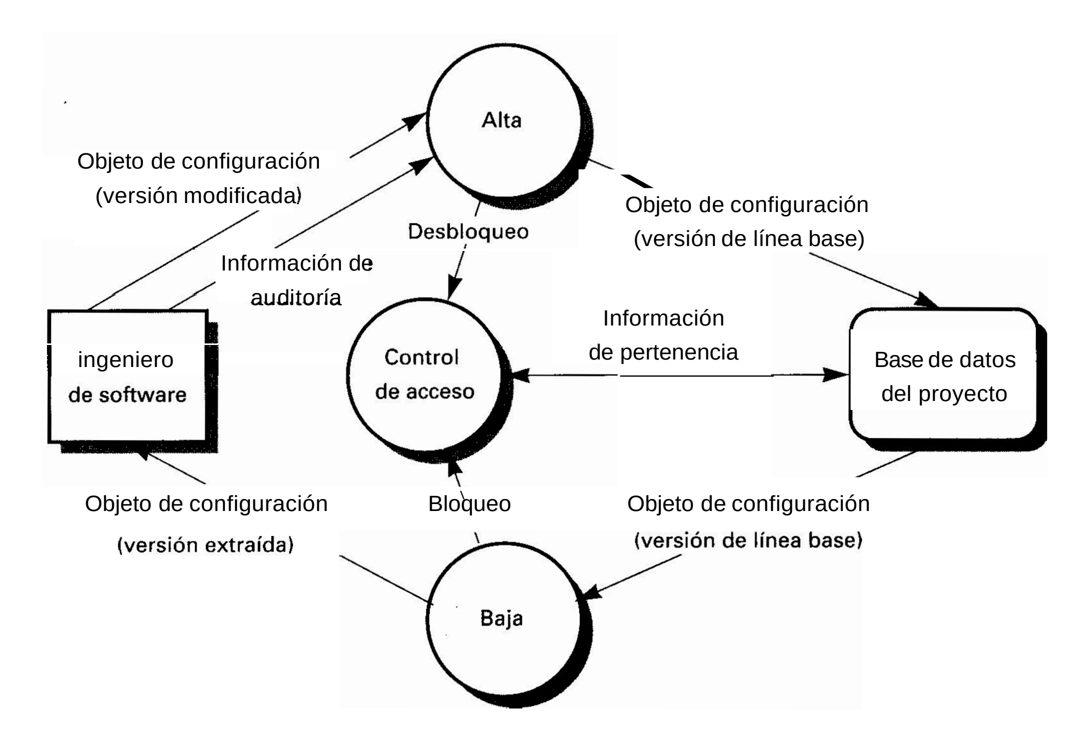


FIGURA 9.6. Control de acceso **y** de sincronización.



La confusión conduce a los errores - algunas de ellas muy serias —. Los controles de acceso y de sincronización evitan la confusión. Implemente ambos, incluso si su enfoque tiene que ser simplificado para adaptarlo a su cultura de desarrollo.

Los procesos de «alta» y «baja» implementan dos elementos importantes del control de cambios --control de acceso y control de sincronización—. El *control de acceso* gobierna los derechos de los ingenieros de software a acceder y modificar objetos de configuración concretos. El *control de sincronización* asegura que los cambios en paralelo, realizados por personas diferentes, no se sobreescriben mutuamente [HAR89].

El flujo de control de acceso y de sincronización están esquemáticamente ilustrados en la Figura 9.6. De acuerdo con una petición de cambio aprobada y una OCI, un ingeniero de software *da de baja* a un objeto de configuración. Una función de control de acceso comprueba que el ingeniero tiene autoridad para dar de baja el objeto, y el control de sincronización *bloquea* el objeto en la base de datos del proyecto, de forma que no se puedan hacer más actualizaciones hasta que se haya reemplazado con la nueva versión. Fíjese que se pueden dar de baja otras copias, pero no se podrán hacer otras actualizaciones. El ingeniero de software modifica una copia del objeto de línea base, denominada *versión extraída*. Tras la **SQA** y la prueba apropiada, se *da de alta* la versión modificada del objeto y se desbloquea el nuevo objeto de línea base.

Puede que algunos lectores empiecen a sentirse incómodos con el nivel de burocracia que implica el proceso anterior. Esta sensación es normal. Sin la protección adecuada, el control de cambios puede ralentizar el progreso y crear un papeleo innecesario. La mayoría de los desarrolladores de software que disponen de mecanismos de control de cambios (desgraciadamente la mayoría no tienen ninguno) han creado varios niveles de control para evitar los problemas mencionados anteriormente.



Opte par un poco más de control de cambio del que pensaba necesitar en un principia. Es probable que mucha puedo ser la cantidad apropiada.

Antes de que un ECS se convierta en una línea base, sólo es necesario aplicar *un control de cambios informal*. El que haya desarrollado el ECS en cuestión podrá hacer cualquier cambio justificado por el proyecto **y** por los requisitos técnicos (siempre que los cambios no impacten en otros requisitos del sistema más amplios que queden fuera del ámbito de trabajo del encargado del desarrollo). Una vez que el objeto ha pasado la revisión técnica formal y ha sido aprobado, se crea la línea base. Una vez que el **ECS** se convierte en una línea base, aparece el *control de cambios a nivel de proyecto*. Ahora, para hacer un cambio, el encargado del desarrollo debe recibir la aprobación del gestor del proyecto (si el cambio es «local») o de la ACC si el cambio impacta en otros ECSs. En algunos casos, se dispensa de generar formalmente las peticiones de cambio, los informes de cambio y las OCI. Sin embargo, hay que hacer una evaluación de cada cambio así como un seguimiento y revisión de los mismos.

Cuando se distribuye el producto de software a los clientes, se instituye el *control de cambios formal*. El procedimiento de control de cambios formal es el que aparece definido en la Figura 9.5.

La autoridad de control de cambios (ACC) desempeña un papel activo en el segundo y tercer nivel de control. Dependiendo del tamaño y de las características del proyecto de software, la ACC puede estar compuesta por una persona —el gestor del proyecto— o por varias personas (por ejemplo, representantes del software, del hardware, de la ingeniería de las bases de datos, del soporte o del departamento comercial, etc.). El papel de la ACC es el de tener una visión general, o sea, evaluar el impacto del cambio fuera del ECS en cuestión. ¿Cómo impactará el cambio en el hardware? ¿Cómo impactará en el rendimiento? ¿Cómo alterará el cambio la percepción del cliente sobre el producto? ¿Cómo afectará el cambio a la calidad y a la fiabilidad? La ACC se plantea estas y otras muchas cuestiones.



El cambio es inevitable, excepto para las máquinas de venta.
Bumper Sticker

9.6 AUDITORÍA DE LA CONFIGURACIÓN

La identificación, el control de versiones y el control de cambios ayudan al equipo de desarrollo de software a mantener un orden que, de otro modo, llevaría a una situación caótica y sin salida. Sin embargo, incluso los mecanismos más correctos de control de cambios hacen un seguimiento al cambio sólo hasta que se ha generado la OCI. ¿Cómo podemos asegurar que el cambio se ha implementado correctamente? La respuesta es doble: (1) revisiones técnicas formales y (2) auditorías de configuración del software.

Las revisiones técnicas formales (presentadas en detalle en el Capítulo 8) se centran en la corrección técnica del elemento de configuración que ha sido modificado. Los revisores evalúan el ECS para determinar la con-

sistencia con otros ECSs, las omisiones o los posibles efectos secundarios. Se debe llevar a cabo una revisión técnica formal para cualquier cambio que no sea trivial.

Una *auditoría de configuración del software* complementa la revisión técnica formal al comprobar características que generalmente no tiene en cuenta la revisión. La auditoría se plantea y responde las siguientes preguntas:

¿ Cuáles son las principales preguntas que hacemos en una auditoría de configuración?

1. ¿Se ha hecho el cambio especificado en la OCI? ¿Se han incorporado modificaciones adicionales?
2. ¿Se ha llevado a cabo una revisión técnica formal para evaluar la corrección técnica?
3. ¿Se ha seguido el proceso del software y se han aplicado adecuadamente los estándares de ingeniería del software?
4. ¿Se han «resaltado» los cambios en el ECS? ¿Se han especificado la fecha del cambio y el autor? ¿Reflejan los cambios los atributos del objeto de Configuración?
5. ¿Se han seguido procedimientos de GCS para señalar el cambio, registrarlo y divulgarlo?
6. ¿Se han actualizado adecuadamente todos los ECSs relacionados?

En algunos casos, las preguntas de auditoría se incluyen en la revisión técnica formal. Sin embargo, cuando la GCS es una actividad formal, la auditoría de la GCS se lleva a cabo independientemente por el grupo de garantía de calidad.

9.7 INFORMES DE ESTADO

La *generación de informes de estado de la configuración* (a veces denominada *contabilidad de estado)* es una tarea de GCS que responde a las siguientes preguntas: (1) ¿Qué pasó? (2) ¿Quién lo hizo? *(3)* ¿Cuándo pasó? (4) ¿Qué más se vio afectado?

En la Figura 9.5 se ilustra el flujo de información para la generación de *informes de estado de la configuración (IEC)*. Cada vez que se asigna una nueva identificación a un ECS o una identificación actualizada se genera una entrada en el IEC. Cada vez que la ACC aprueba un cambio (o sea, se expide una OCI), se genera una entrada en el IEC. Cada vez que se lleva a cabo una auditoría de configuración, los resultados aparecen como parte de una tarea de generación de un IEC. La salida del IEC se puede situar en una base de datos interactiva [TAY85] de forma que los encargados del desarrollo o del mantenimiento del software puedan acceder a la información de cambios por categorías clave. Además, se genera un IEC regularmente con intención de mantener a los gestores y a los profesionales al tanto de los cambios importantes.



Desarrolle una «lista que se necesita conocer» para cada ECS y manténgala actualizada. Cuando se realice un cambio, asegúrese que se informa a todos los que están en la listo.

La generación de informes de estado de la configuración desempeña un papel vital en el éxito del proyecto de desarrollo de software. Cuando aparece involucrada mucha gente es muy fácil que se dé el síndrome de que «la mano izquierda ignora lo que hace la mano derecha». Puede que dos programadores intenten modificar el mismo ECS con intenciones diferentes y conflictivas. Un equipo de ingeniería del software puede emplear meses de esfuerzo en construir un software a partir de unas especificaciones de hardware obsoletas. Puede que la persona que descubra los efectos secundarios senos de un cambio propuesto no esté enterada de que el cambio se está realizando. El IEC ayuda a eliminar esos problemas, mejorando la comunicación entre todas las personas involucradas.

RESUMEN

La gestión de configuración del software es una actividad de protección que se aplica a lo largo de todo el proceso del software. La GCS identifica, controla, audita e informa de las modificaciones que invariablemente se dan al desarrollar el software una vez que ha sido distribuido a los clientes. Cualquier información producida como parte de la ingeniería del software se convierte en parte de una configuración del software. La configuración se organiza de tal forma que sea posible un control organizado de los cambios.

La configuración del software está compuesta por un conjunto de objetos interrelacionados, también denominados elementos de configuración del software, que se producen como resultado de alguna actividad de ingeniería del software. Además de los documentos, los programas y los datos, también se puede poner bajo control de configuración el entorno de desarrollo utilizado para crear el software.

Una vez que se ha desarrollado y revisado un objeto de configuración, se convierte en una línea base. Los

cambios sobre un objeto de línea base conducen a la creación de una nueva versión del objeto. La evolución de un programa se puede seguir examinando el historial de las revisiones de todos los objetos de su configuración. Los objetos básicos y los compuestos forman un asociación de objetos que refleja las variantes y las versiones creadas. El control de versiones es un conjunto de procedimientos y herramientas que se usan para gestionar el uso de los objetos.

El control de cambios es una actividad procedimental que asegura la calidad y la consistencia a medida que se realizan cambios en los objetos de la configuración. El proceso de control de cambios comienza con una petición de cambio, lleva a una decisión de proseguir o no con el cambio y culmina con una actualización controlada del ECS que se ha de cambiar.

La auditoría de configuración es una actividad de **SQA** que ayuda a asegurar que se mantiene la calidad durante la realización de los cambios. Los informes de estado proporcionan información sobre cada cambio a aquellos que tienen que estar informados.

REFERENCIAS

[ADA89] Adams, E., M. Honda y T. Miller, «Object Management in a CASE Environment», *Proc. 11 th Intl. Conf Software Engineering*, IEEE, Pittsburg, PA, Mayo 1989, pp. 154-163.

[BAC98] Bach, J., «The Highs and Lows of Change Control», *Computer*, vol. 31, n.º 8, Agosto 1998, pp. 113-115.

[BER80] Bersoff, E.H., V.D. Henderson y S. G. Siegel, *Software Configuration Management*, Prentice-Hall, 1980.

[BRY80] Bryan, W., C. Chadboume, y S. Siegel, *Software Configuration Management*, IEEE Computer Society Press, 1980.

[CHO89] Choi, S. C., y W. Scacchi, «Assuring the Correctness of a Configured Software Description», *Proc. 2nd Intl. Workshop on Software Conftguration Management*, ACM, Princeton, NJ, Octubre 1989, pp. 66-75.

[CLE89] Clemm, G. M., «Replacing Version Control with Job Control», *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, Octubre 1989, pp. 162-169.

[GUS89] Gustavsson, A., «Maintaining the Evaluation of Software Objects in an Integrated Environment», *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, Octubre 1989, pp. 1 14-117.

[HAR89] Harter, R., «Configuration Management», HP *Professional*, **vol. 3**, n.º 6, Junio 1989.

[IEE94] *Software Engineering Standards*, edición de 1994, IEEE Computer Society, 1994.

[LIE89] Lie, A. *et al.*, «Change Oriented Versioning in a Software Engineeering Database», *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, Octubre 1989, pp. 56-65

[NAR87] Narayanaswamy, **K.**, y W. Scacchi, «Maintatning Configurations of Evolving Software Systems», *IEEE Trans. Software Engineering*, **vol.** SE-13, n.º 3, Marzo 1987, pp. 324-334.

[REI89] Reichenberger, C., «Orthogonal Version Management», *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, Octubre 1989, pp. 137-140.

[ROC75] Rochkind, M., «The Source Code Control System», *IEEE Trans. Software Engineering*, *vol.* SE-1, n.º 4, Diciembre 1975, pp. 364-370.

[TAY85] Taylor, B., «A Database Approach to Configuration Management for Large Projects», *Proc. Conf. Software Maintenance-1985*, IEEE, Noviembre 1985, pp. 15-23.

[TIC82] Tichy, W. F., «Design, Implementation and Evaluation of a Revision Control System», *Proc. 6th Intl. Conf. Software Engineering*, IEEE, Tokyo, Septiembre 1982, pp. 58-67.

PROBLEMAS Y PUNTOS A CONSIDERAR

9.1. ¿Por qué es cierta la primera ley de la ingeniería de sistemas? ¿Cómo afecta a nuestra percepción de los paradigmas de la ingeniería del software?

9.2. Exponga las razones de la existencia de líneas base con sus propias palabras.

9.3. Asuma que es el gestor de un pequeño proyecto. ¿Qué líneas base definiría para el proyecto y cómo las controlaría?

9.4. Diseñe un sistema de base de datos que permita a un ingeniero del software guardar, obtener referencias de forma cruzada, buscar, actualizar, cambiar, etc., todos los elementos de la configuración del software importantes. ¿Cómo manejaría la base de datos de diferentes versiones de un mismo programa? ¿Se manejaría de forma diferente el código fuente que la documentación? ¿Cómo se evitaría que dos programadores hicieran cambios diferentes sobre el mismo ECS al mismo tiempo?

9.5. Investigue un poco sobre bases de datos orientadas a objetos y escriba un artículo que describa cómo se podrían usar en el contexto de la GCS.

9.6. Utilice un modelo E-R (Capítulo 12) para describir las interrelaciones entre los ECS (objetos) de la Sección 9.1.2.

9.7. Investigue sobre herramientas de GCS existentes y describa cómo implementan el control de versiones, de cambios y de objetos de configuración.

9.8. Las relaciones «parte-de» e «interrelacionado» representan relaciones sencillas entre los objetos de configuración. Describa cinco relaciones adicionales que pudieran ser Útiles en el contexto de la base de datos del proyecto.

9.9. Investigue sobre una herramienta de GCS existente y describa cómo implementa los mecanismos de control de versiones. Alternativamente, lea dos o tres de los artículos a los que se hace referencia en este capítulo e investigue en las estructuras de datos y los mecanismos que se usan para el control de versiones.

9.10. Utilizando la Figura 9.5 como guía, desarrolle un esquema de trabajo más detallado aún para el control de cambios. Describa el papel de la ACC y sugiera formatos para la petición de cambio, el informe de cambios e IEC.

9.11. Desarrolle una lista de comprobaciones que se pueda utilizar en las auditorías de configuración.

9.12. ¿Cuál es la diferencia entre una auditoria de GCS y una revisión técnica formal? ¿Se pueden juntar sus funciones en una sola revisión? ¿Cuáles son los pros y los contras?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Uno de los pocos libros escritos sobre la GCS en los últimos años lo realizaron Brown et al. (*AntiPatterns and Patterns in Software Configuration Managernent,* Wiley, 1989).

Los autores tratan las cosas que no hay que hacer (antipatrones) cuando se implementa un proceso de GCS y entonces consideran sus remedios.

Lyon(*Practical CM: Best Configuration Management Practices for the 21st Century,* Raven Publishing, 1999) y Mikkelsen y Pherigo *(Practica1 Software Configuration Management: The Latenight Developer's Handhook,* Allyn & Bacon, 1997) proporcionan tutoriales practicos importantes de GCS. Ben-Menachem (*Software Configuration Management Guidebook,* McGraw-Hill, 1994), Vacca (*Implementing a Successful. Configuration. Change. Management. Program,* I.S. Management Group, 1993), y Ayer y Patrinnostro *(Software Configuration Management,* McGrawHill, 1992) presentan correctas visiones para aquellos que todavia no se han introducido en la materia. Berlack *(Software Configuration Management,* Wiley, 1992, presenta una estudio Útil de conceptos de la GCS, haciendo incapié en la importancia del diccionario de datos (repository) y de las herramientas en la gestion del cambio. Babich [BAB86] es un tratado abreviado, aunque eficaz, de temas practicos sobre la gestión de configuración del software.

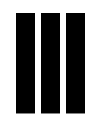
Buckley *(Implementing Configuration Managernent,* IEEE Computer Society Press, 1993) estudia enfoques de gestión de configuración para todos los elementos de un sistema (hardware, software y firmware con unos detallados tratamientos de las principales actividades de GC). Rawlings *(SCM* for *Network Development Environments,* McGraw-Hill, 1994) es el primer libro de GCS en tratar el asunto con un énfasis específico en el desarrollo de software en un entorno de red. Whitgift *(Methods and Tools for Software Configuration Management,* Wiley, 1991) contiene una razonable cobertura de todos los temas importantes de GCS, pero se distingue por su estudio del diccionario de datos y tratamiento de aspectos de entornos CASE. Arnold y Bohner *(Software Change lmpact Analysis,* IEEE Computer Society Press, 1996) han editado una antología que estudia cómo analizar el impacto del cambio dentro de sistemas complejos basados en software.

Puesto que la GCS identifica y controla documentos de ingenieria del software, los libros de Nagle *(Handbook for Preparing Engineering Documents: From Concept to Completion,* IEEE, 1996), Watts *(Engineering Documentation Control Handbook: Configuration Management for Industry,* Noyes Publication, 1993). Ayer y Patnnnostro (*Documenting the Software Process,* McGraw-Hill, 1992) proporciona un complemento con textos mas centrados en la GCS. La edición de Marzo de 1999 de *Crosstalk* contiene vanos artículos Útiles de la GCS.

En Internet están disponibles una gran variedad de fuentes de información relacionadas con temas de gestión **y** de software. Se puede encontrar una lista actualizada con referencias a sitios (páginas) web que son relevantes para el Software en **http://www.pressman5.com.**

.

PARTE



MÉTODOS CONVENCIONALES PARA LA INGENIERÍA DEL SOFTWARE

EN esta parte de *Ingeniería del Software Un Enfoque Práctico* consideramos los conceptos técnicos, métodos y mediciones que son aplicables al análisis, diseño y pruebas del software. En los capítulos siguientes, aprenderás las respuestas a las siguientes cuestiones:

- ¿Cómo se define el software dentro del contexto de **un** gran sistema y **qué** papel juega la ingeniería de sistemas?
- ¿Cuáles son los principios y conceptos básicos que se aplican al análisis de requisitos del software?
- ¿Que es el análísis estructurado y cómo sus diferentes modelos te permiten entender las funciones, datos y comportamientos?
- ¿Cuáles son los conceptos básicos y **los** principios que ***se* aplican** en la actividad de diseño del software?
- ¿Cómo se realiza el diseño de los modelos de datos; arquitectura, interfaces y componentes?
- ¿Cuáles son los conceptos básicos, principios y estrategias que se aplican a las pruebas del software?
- ;Cómo se utilizan los métodos de prueba de caja negra y caja blanca para diseñar eficientes casos de prueba?
- ¿Qué métricas técnicas están disponibles para valorar la calidad de los modelos de análisis y diseño, código fuente y casos de prueba?

Una vez que estas cuestiones sean contestadas, comprenderás cómo construir software utilizando un enfoque disciplinado de ingeniería.

.

CAPÍTULO

10 INGENIERÍA DE SISTEMAS

HACE casi 500 años, Maquiavelo dijo: «... no hay nada más difícil de llevar a cabo, más peligroso de realizar o de éxito más incierto que tomar el liderazgo en la introducción de un nuevo orden de cosas». Durante los Últimos 50 años, los sistemas basados en computadora han introducido un nuevo orden. Aunque la tecnología ha conseguido grandes avances desde que habló Maquiavelo, sus palabras siguen sonando a verdad.

La ingeniería del software aparece como consecuencia de un proceso denominado *ingeniería **de** sistemas.* En lugar de centrarse únicamente en el software, la ingeniería de sistemas se centra en diversos elementos, analizando, diseñando y organizando esos elementos en un sistema que pueden ser un producto, un servicio o una tecnología para la transformación de información o control de información.

El proceso de ingeniería de sistemas es denominado *ingeniería **de** procesos **de** negocio* cuando el contexto del trabajo de ingeniería se enfoca a una empresa. Cuando hay que construir un producto, el proceso se denomina *ingeniería **de** producto.*

Tanto la ingeniería de proceso de negocio como la de producto intentan poner orden al desarrollo de sistemas basados en computadoras. Aunque cada una se aplica en un dominio de aplicación diferente, ambas intentan poner al software en su contexto.

VISTAZO RÁPIDO

¿Qué es? Antes de que el software se pueda construir, el «sistema» en el que residirá se debe comprender. Para lograrlo, se deben definir los objetivos generales del sistema; se debe identificar el papel del hardware, software, personas, bases de datos, procedimientos y otros elementos del sistema; y los requerimientos operacionales deben ser identificados; analizados, especificados, modelizados, validados y gestionados. Estas actividades son la base de la ingeniería de sistemas.

¿Quién lo hace? Un ingeniero de sistemas trabaja para comprender los requisitos del sistema en colaboración con el cliente, los futuros usuarios y otras partes interesadas.

¿Por qué es importante? Hay un viejo dicho que dice que «los árboles no dejan ver el bosque». En este contexto, el «bosque» es el sistema, y los árboles son los elementos tecnológicos (incluido el software) que son requeridos para realizar el sistema. El empeño en construir elementos tecnológicos, antes de comprender el sistema, lleva a cometer errores que desagradarán a los clientes.

¿Cuáles son los pasos? Los objetivos y los requisitos operacionales de mayor detalle son identificados gracias a la información facilitada por el cliente. Los requisitos son analizados para valorar su claridad, completitud y consistencia. Una especificación, *incorporada a un modelo del sistema,* se crea y valida posteriormente por los clientes y las partes interesadas. Finalmente, los requisitos del sistema son gestionados para asegurar que los cambios se controlan adecuadamente.

¿Cuál es el producto obtenido? Se debe obtener una correcta representación del sistema como consecuencia de la ingeniería de sistemas. Se puede realizar a través de un prototipo, una especificación o, incluso, un modelo simbólico, debiendo comunicar la operativa, la funcionalidad y las características de comportamiento del sistema que se va a construir e incorporarlo dentro de la arquitectura del sistema.

¿Cómo puedo estar seguro de que lo he hecho correctamente? El producto obtenido, a través de la aplicación de la ingeniería de sistemas, debe ser revisado para determinar su claridad, completitud y consistencia. Es importante que los cambios en los requisitos de un sistema sean gestionados utilizando métodos sólidos de GCS (Capítulo 9).

Es decir, tanto la ingeniería de procesos de negocio[1] como la de producto trabajan para asignar un papel al software de computadora y para establecer los enlaces que unen al software con otros elementos de un sistema basado en computadora.

En este capítulo, profundizamos en las necesidades de gestión y en las actividades específicas del proceso que permitan asegurar una organización del software que consiga resultados satisfactorios en el tiempo fijado y por el método definido.

[1] En realidad, el término *ingeniería de sistemas* se emplea a menudo en este contexto. Sin embargo, en este libro, el término «ingeniería de sistemas» es genérico y se usa para abarcar **a** la ingeniería de proceso de negocio y a la ingeniería de producto.

10.1 SISTEMAS BASADOS EN COMPUTADORA

La palabra *sistema* es posiblemente el término más usado y abusado del léxico técnico. Hablamos de sistemas políticos y de sistemas educativos, de sistemas de aviación y de sistemas de fabricación, de sistemas bancarios y de sistemas de locomoción. La palabra no nos dice gran cosa. Usamos el adjetivo para describir el sistema y para entender el contexto en que se emplea. El *diccionario Webster* define sistema como:

> 1. un conjunto o disposición de cosas relacionadas de manera que forman una unidad o un todo orgánico; **2.** un conjunto de hechos, principios, reglas, etc., clasificadas y dispuestas de manera ordenada mostrando un plan lógico de unión de las partes; **3.** un método o plan de clasificación o disposición; **4.** una manera establecida de hacer algo; método; procedimiento...

Se proporcionan cinco definiciones más en el diccionario, pero no se sugiere un sinónimo preciso. Sistema es una palabra especial.

Tomando prestada la definición del diccionario Webster, definimos un *sistema basado en computadora* como:

> Un conjunto o disposición de elementos que están organizados para realizar un objetivo predeñnido procesando información.

El objetivo puede ser soportar alguna función de negocio o desarrollar un producto que pueda venderse para generar beneficios. Para conseguir el objetivo, un sistema basado en computadora hace uso de varios elementos del sistema:

> **Software. Programas** de computadora, estructuras de datos y su documentación que sirven para hacer efectivo el método lógico, procedimiento o control requerido.
>
> **Hardware.** Dispositivos electrónicos que proporcionan capacidad de cálculo, dispositivos de interconexión (por ejemplo, conmutadores de red, dispositivos de telecomunicación) y dispositivos electromecánicos (por ejemplo, sensores, motores, bombas) que proporcionan una función externa, del mundo real.
>
> **Personas.** Usuarios y operadores del hardware y software.
>
> **Documentación.** Manuales, formularios y otra información descriptiva que plasma el empleo y/o funcionamiento del sistema.
>
> **Procedimientos. Los** pasos que definen el empleo específico de cada elemento del sistema o el contexto procedimental en que reside el sistema.



No esté atraído por hablar de un planteamiento «centrado en el software». Comience por considerar todos los elementos de un sistema antes de concentrarse en el software.

Los elementos se combinan de varias maneras para transformar la información. Por ejemplo, un departamento de marketing transforma la información bruta de ventas en un perfil del típico comprador del producto; un robot transforma un archivo de órdenes, que contiene instrucciones específicas, en un conjunto de señales de control que provocan alguna acción física específica. Tanto la creación de un sistema de información para asesorar a un departamento de marketing, como el software de control para el robot, requieren de la ingeniería de sistemas.

Una característica complicada de los sistemas basados en computadora es que los elementos que componen un sistema pueden también representar un macroelemento de un sistema aún más grande. El *macroelemento* es un sistema basado en computadora que es parte de un sistema más grande basado en computadora. Por ejemplo, consideremos un «sistema de automatización de una fábrica» que es esencialmente una jerarquía de sistemas. En el nivel inferior de la jerarquía tenemos una máquina de control numérico, robots **y** dispositivos de entrada de información. Cada uno es un sistema basado en computadora por derecho propio. Los elementos de la máquina de control numérico incluyen hardware electrónico y electromecánico (por ejemplo, procesador y memoria, motores, sensores); software (para comunicaciones, control de la máquina e interpolación); personas (el operador de la máquina); una base de datos (el programa CN almacenado); documentación **y** procedimientos. Se podría aplicar una descomposición similar a los dispositivos de entrada de información y al robot. Todos son sistemas basados en computadora.



Los sistemas complejos son actualmente una jerarquía de macro elementos que son sistemas en sí mismos.

En el siguiente nivel de la jerarquía, se define una célula de fabricación. La célula de fabricación es un sistema basado en computadora que puede tener elementos propios (por ejemplo, computadoras, fijaciones mecánicas) y también integra los macroelementos que hemos denominado máquina de control numérico, robot y dispositivo de entrada de información.

Para resumir, la célula de fabricación y sus macroelementos están compuestos de elementos del sistema con las etiquetas genéricas: software, hardware, personas, base de datos, procedimientos y documentación. En algunos casos, los macroelementos pueden compartir un elemento genérico. Por ejemplo, el robot y la máquina CN podrían ser manejadas por el mismo operador (el elemento personas). En otros casos, los elementos genéricos son exclusivos de un sistema.

El papel del ingeniero de sistemas es definir los elementos de un sistema específico basado en computadora en el contexto de la jerarquía global de sistemas (macroelementos).

En las siguientes secciones, examinamos las tareas que constituyen la ingeniería de sistemas de computadoras.

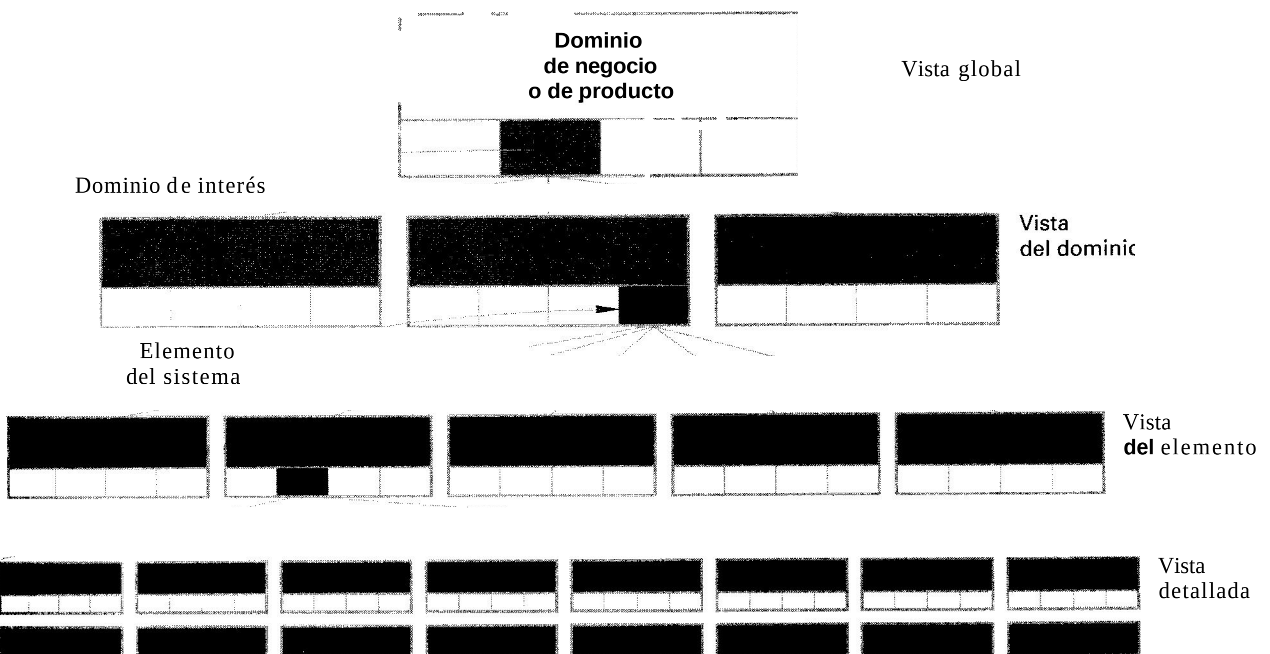


FIGURA 10.1. La jerarquía de la ingeniería de sistemas de computadora.

10.2 LA JERARQUÍA DE LA INGENIERÍA DE SISTEMAS

Independientemente del dominio de enfoque, la ingeniería de sistemas comprende una colección de métodos para navegar de arriba abajo y de abajo arriba en la jerarquía ilustrada en la Figura 10.1. El proceso de la ingeniería de sistemas empieza normalmente con una «visión global». Es decir, se examina el dominio entero del negocio o del producto para asegurarse de que se puede establecer el contexto de negocio o tecnológico apropiado. La visión global se refina para enfocarse totalmente en un dominio de interés específico. Dentro de un dominio específico, se analiza la necesidad de elementos del sistema (por ejemplo, información, software, hardware, personas). Finalmente, se inicia el análisis, diseño y construcción del elemento del sistema deseado. En la parte alta de la jerarquía se establece un contexto muy amplio y en la parte baja se llevan a cabo actividades técnicas detalladas, realizadas por la disciplina de ingeniería correspondiente (por ejemplo, ingeniería hardware o software)[2].

10.2.1. Modelado del sistema

La ingeniería de sistemas de computadora es un proceso de modelado. Tanto si el punto de mira está en la visión global o en la visión detallada, el ingeniero crea modelos que [MOT92]:

- definan los procesos que satisfagan las necesidades de la visión en consideración;
- representen el comportamiento de los procesos y los supuestos en los que se basa el comportamiento;
- definan explícitamente las entradas exógenas[3] y endógenas de información al modelo;
- representen todos las uniones (incluyendo las salidas) que permitan al ingeniero entender mejor la visión.



tos buenos sistemas de ingeniería comienzan por clarificar el comportamiento de contexto —lo visión global — y progresivamente se van estrechando hasta el nivel de detalle necesario.

Para construir un modelo del sistema, el ingeniero debería considerar algunas restricciones:

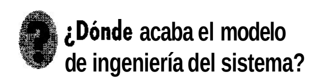
1. *Supuestos* que reducen el número de permutaciones y variaciones posibles, permitiendo así al modelo reflejar el problema de manera razonable. Por ejem-

[2] En algunas situaciones, sin embargo, los ingenieros del sistema deben considerar primero los elementos individuales del sistema y/o los requisitos detallados. Empleando este enfoque, los subsistemas se escriben de abajo a arriba considerando primero los componentes de detalle del subsistema.

[3] Las entradas *exógenas* unen un elemento de una visión dada con otros elementos al mismo o a otros niveles; las entradas *endógenas* unen componentes individuales de un elemento en una visión particular.

plo, considere un producto de representación en tres dimensiones usado por la industria de entretenimiento para crear animaciones realistas. Un dominio del producto permite la representación de formas humanas en 3D. Las entradas a este dominio comprenden la habilidad de introducir movimiento de un «actor» humano vivo, desde vídeo o creando modelos gráficos. El ingeniero del sistema hace ciertos supuestos sobre el rango de movimientos humanos permitidos (por ejemplo, las piernas no pueden enrollarse alrededor del tronco) de manera que puede limitarse el proceso y el rango de entradas.

2. *Simplificaciones* que permiten crear el modelo a tiempo. Para ilustrarlo, considere una compañía de productos de oficina que vende y suministra una amplia variedad de fotocopiadoras, faxes y equipos similares. El ingeniero del sistema está modelando las necesidades de la organización suministradora y está trabajando para entender el flujo de información que engendra una orden de suministro. Aunque una orden de suministro puede generarse desde muchos orígenes, el ingeniero categoriza solamente dos fuentes: demanda interna o petición externa. Esto permite una partición simplificada de entradas necesaria para generar una orden de trabajo.
3. *Limitaciones* que ayudan a delimitar el sistema. Por ejemplo, se está modelando un sistema de aviónica para un avión de próxima generación. Como el avión tendrá un diseño de dos motores, todos los dominios de supervisión de la propulsión se modelarán para albergar un máximo de dos motores y sus sistemas redundantes asociados.



4. *Restricciones* que guían la manera de crear el modelo y el enfoque que se toma al implementar el modelo. Por ejemplo, la infraestructura tecnológica para el sistema de representación en tres dimensiones descrita anteriormente es un solo procesador basado en un Power-PC. La complejidad de cálculo de los problemas deben restringirse para encajar en los límites de proceso impuestos por el procesador.



Un ingeniero del sistema considera los siguientes factores cuando desarrolla soluciones alternativas: planteamientos, simplificaciones, limitaciones, restricciones y preferencias de los clientes.

5. *Preferencias* que indican la arquitectura preferida para todos los datos, funciones y tecnología. La solución preferida entra a veces en conflicto con otros factores restrictivos. Aunque la satisfacción del cliente es a menudo tomada en cuenta hasta el punto de realizar su enfoque preferido.

El modelo de sistema resultante (desde cualquier visión) puede reclamar una solución completamente automática, semiautomática o un enfoque manual. De hecho, es posible a menudo caracterizar modelos de cada tipo que sirven de soluciones alternativas para el problema que tenemos entre manos. En esencia, el ingeniero del sistema modifica simplemente la influencia relativa de los diferentes elementos del sistema (personas, hardware, software) para crear modelos de cada tipo.

10.2.2. Simulación del sistema

En los años 60, R.M. Graham [GRA69] hizo un comentario crítico sobre la manera en que se construían los sistemas basados en computadora: «Construimos sistemas igual que los hermanos Wright construían aviones: construimos todo el sistema, lo empujamos barranco abajo, le dejamos que se estrelle y empezamos de nuevo.» De hecho, para al menos un tipo de sistema —*el sistema reactivo*— lo continuamos haciendo hoy en día.

Muchos sistemas basados en computadora interaccionan con el mundo real de forma reactiva. Es decir, los acontecimientos del mundo real son vigilados por el hardware y el software que componen el sistema, y basándose en esos sucesos, el sistema aplica su control sobre las máquinas, procesos e incluso las personas que motivan los acontecimientos. Los sistemas de tiempo real y sistemas empotrados pertenecen a menudo a la categoría de sistemas reactivos.



Si la capacidad de simulación no está disponible para un sistema reactivo, el riesgo del proyecto se incrementa. Considerar para su utilización un modelo de proceso iterativo que te permita obtener un resultado en una primera iteración y utilizar otras iteraciones para ajustar sus características.

Desgraciadamente, **los** desarrolladores de sistemas reactivos luchan a veces para hacerlos funcionar correctamente. Hasta hace poco, ha sido difícil predecir el rendimiento, la eficacia y el comportamiento de estos sistemas antes de construirlos. Realmente, la construcción de muchos sistemas de tiempo real era una aventura. Las sorpresas (la mayoría desagradables) no se descubrían hasta que el sistema era construido **y** «arrojado colina abajo». Si el sistema se estrellaba debido a un funcionamiento incorrecto, comportamiento inapropiado o escaso rendimiento, cogíamos las piezas y empezábamos de nuevo.

Muchos sistemas de la categoría de los reactivos controlan máquinas y/o procesos (por ejemplo, aerolíneas comerciales o refinerías de petróleo) que deben operar con extrema fiabilidad.

Si el sistema falla, podrían ocurrir pérdidas económicas o humanas significativas. Por este motivo, el enfoque descrito por Graham es penoso y peligroso.

Hoy en día se utilizan herramientas software para el modelado y simulación de sistemas para ayudar a eliminar sorpresas cuando se construyen sistemas reactivos basados en computadora. Estas herramientas se aplican durante el proceso de ingeniería de sistemas, mientras se están especificando las necesidades del hardware, software, bases de datos y de personas. Las herramientas de modelado y simulación capacitan al ingeniero de sistemas para probar una especificación del sistema. En el Capítulo **31** se estudian brevemente los detalles técnicos **y** técnicas especiales de modelado que se emplean para llevar a cabo estas pruebas.



Herramientas CASE
Modelado y Simulación

10.3 INGENIERÍA DE PROCESO DE NEGOCIO: UNA VISIÓN GENERAL

El objetivo de la ***ingeniería de proceso de negocio (IPN)*** es definir arquitecturas que permitan a las empresas emplear la información eficazmente. Michael Guttman [GUT99] describe el desafio cuando dice:

> **El** actual entorno computacional consiste en un poder de computación distribuido en toda la empresa con múltiples unidades diferentes de procesamiento, dividido y configurado por una amplía variedad de tareas, Nuevos planteamientos como la computación cliente-servidor, procesamiento distribuido, el trabajo en red (por nombrar algunos de **los** términos más sobreusados) permiten gestionar las demandas aportando mayor funcionalidad y flexibilidad.
>
> Sin embargo, el coste de estos cambios es ampliamente discutido por la organizaciones de ***TI (Tecnologías de la Información)*** que deben soportar esta políglota configuración. Hoy, cada organización de **TI** debe favorecer la integración de **sus** sistemas. Debe diseñar, implementar y soportar su propia configuración de recursos de computación heterogénea, distribuidos lógica y geográficamente por toda la empresa, conectándola a través de un esquema apropiado para el trabajo en red.
>
> Por otra parte, esta configuración debe ser diseñada para cambios continuos, desigualmente localizados en la empresa, debido a cambios en requisitos del negocio y en las propias tecnologías. Estos diversos e incrementales cambios deben ser coordinados a través del entorno distribuido, consistente en hardware y software suministrado por decenas, cuando no cientos, de vendedores. Por supuesto, esperamos que estos cambios los incorporemos sin ruptura con la operativa habitual permitiendo además ampliar la operativa.

Cuando hablamos de una visión general de las necesidades de tecnología de información de una compañía, existen pequeñas incertidumbres que son planteadas a la ingeniería de sistemas. La ingeniería de proceso de negocio es un acercamiento para crear un plan general para implementar la arquitectura de computación [SPE93].



Tres arquitecturas diferentes son desarrolladas durante la IPN: la arquitectura de datos, la arquitectura de aplicación y la infraestructura tecnológica.

Se deben analizar y diseñar tres arquitecturas diferentes dentro del contexto de objetivos y metas de negocio:

- arquitectura de datos
- arquitectura de aplicaciones
- infraestructura de la tecnología

Referencia cruzada

Los objetos de datos son tratados en detalle en el Capítulo 12.

La arquitectura de datos proporciona una estructura para las necesidades de información de un negocio o de una función de negocio. Los ladrillos de la arquitectura son los objetos de datos que emplea la empresa. Un objeto de datos contiene un conjunto de atributos que definen aspectos, cualidades, características o descriptor de los datos que han sido descritos. Por ejemplo, un ingeniero de la información puede definir el objeto de datos: cliente. Para describir más en detalle al cliente, se definen los siguientes atributos:

Objeto: Cliente
Atributos:
nombre
nombre de la compañía
clasificación del trabajo y autoridad en compra
dirección comercial e información de contacto
producto(s) ***de interés***
compra(s) ***anteriores***
fecha de último contacto
situación del contacto

Una vez definido el conjunto de datos, se identifican sus relaciones. Una ***relación*** indica como los objetos están conectados. Como ejemplo, considerar los objetos: **cliente** y **producto A.** Los dos objetos pueden conectarse por la relación ***compra;*** es decir, un cliente compra el producto **A** o el producto **A** es comprado por un cliente. Los objetos de datos (pueden existir cientos o miles

para una actividad de negocio importante) fluyen entre las funciones de negocio, están organizados dentro de una base de datos y se transforman para proveer información que sirva a las necesidades del negocio.

La arquitectura de aplicación comprende aquellos elementos de un sistema que transforman objetos dentro de la arquitectura de datos por algún propósito del negocio. En el contexto de este libro, consideramos normalmente que la arquitectura de aplicación es el sistema de programas (software) que realiza esta transformación. Sin embargo, en un contexto más amplio, la arquitectura de aplicación podría incorporar el papel de las personas (por ejemplo, cliente/servidor) que ha sido diseñado para implementar estas tecnologías.

Referencia cruzada

El detalle sobre la arquitectura del software es presentado en el Capítulo 14.

La infraestructura tecnológica proporciona el fundamento de las arquitecturas de datos y de aplicaciones. La infraestructura comprende el hardware y el software empleados para dar soporte a las aplicaciones y datos. Esto incluye computadoras y redes de computadora, enlaces de telecomunicaciones, tecnologías de almacenamiento y la arquitectura (por ejemplo, cliente/servidor) diseñada para implementar estas tecnologías.

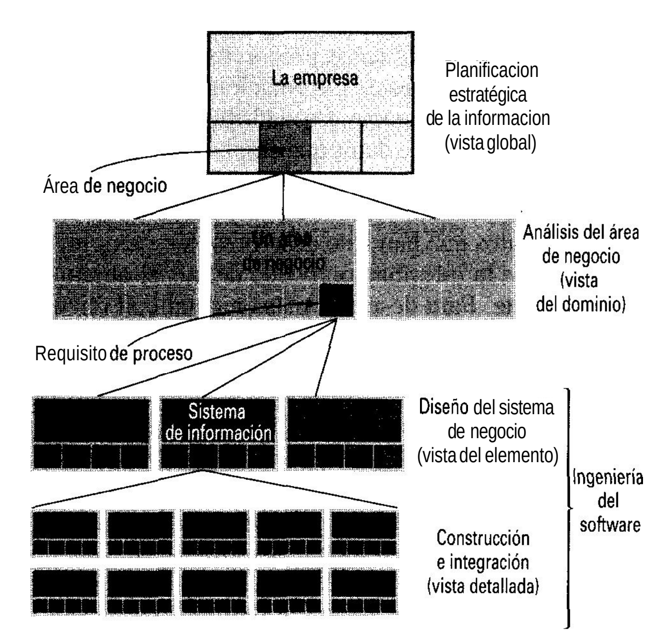


FIGURA 10.2. La jerarquía de la ingeniería de procesos.

Para modelar las arquitecturas de sistema descritas anteriormente, se define una jerarquía de actividades de ingeniería de la información. Como muestra la Figura 10.2, la visión global se consigue a través de la *planificación de la estrategia de información (PEI)*. La PEI ve todo el negocio como una entidad y aisla los dominios del negocio (por ejemplo, ingeniería, fabricación, marketing, finanzas, ventas) importantes para la totalidad de la empresa. La PEI define los objetos de datos visibles a nivel empresa, sus relaciones y cómo fluyen entre los dominios del negocio [MAR90].

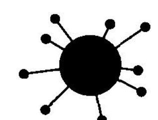


*Iomo ingeniero del **software**, no debes profundizar en **PEI** ni en el **AAN**. **No** obstante, si no está cloro que estas **actividades hayan** sido **realizadas**, informe a su superior que el riesgo **del proyecto** es muy alto.*

La vista del dominio se trata con una actividad IPN denominada *análisis del área de negocio (AAN)*. Hares [HAR93] describe AAN de la siguiente manera:

> El AAN se ocupa de identificar en detalle la información **(en** la forma de tipos de entidad [objeto datos]) y los requisitos de las funciones (en la forma de procesos) de áreas de negocio seleccionadas [dominios] identificadas durante el PEI, averiguando sus interacciones (en forma de matrices). Se ocupa solamente de especificar **qué** se requiere en un área de negocio.

A medida que el ingeniero de información comienza el AAN, el enfoque se estrecha hacia un dominio del negocio específico. El AAN ve el área del negocio como una entidad y aísla las funciones de negocio y procedimientos que permiten al área del negocio lograr sus objetivos y metas. El *AAN*, al igual que el PEI, define objetos de datos, sus relaciones y cómo fluye la información. Pero a este nivel, estas características están delimitadas por el área de negocio que se está analizando. El resultado de AAN es aislar las áreas de oportunidad en las que los sistemas de información pueden prestar soporte al área de negocio.



Ingeniería de Procesos de Negocio

Una vez que se ha aislado un sistema de información para un desarrollo posterior, la IPN hace una transición a la ingeniería del software. Invocando la fase del *diseño de sistema de negocio (DSN)*, se modelan los requisitos básicos de un sistema de información específico y estos requisitos se traducen en arquitectura de datos, arquitectura de aplicación e infraestructura tecnológica.

El paso final de la IPN *(construcción e integración, C&I)* se centra en los detalles de la implementación. La arquitectura e infraestructura se implementan construyendo una base de datos apropiada y estructuras internas de datos, mediante la construcción de aplicaciones que están constituidas por programas, y seleccionando los elementos apropiados de una infraestructura tecnológica para dar soporte al diseño creado durante el DSN. Cada uno de estos componentes del sistema debe integrarse para formar una aplicación o sistema de información completo. La actividad de integración también coloca al nuevo sistema de información en el contexto del área de negocio, realizando todo el entrenamiento de usuario y soporte logístico para conseguir una suave transición.

10.4 INGENIERÍA DE PRODUCTO: UNA VISIÓN GENERAL

La meta de la ingeniería de producto es traducir el deseo de un cliente, de un conjunto de capacidades definidas, a un producto operativo[4]. Para conseguir esta meta, la ingeniería de producto (como la ingeniería de proceso de negocio) debe crear una arquitectura y una infraestructura. La arquitectura comprende cuatro componentes de sistema distintos: software, hardware, datos (bases de datos) y personas. Se establece una infraestructura de soporte e incluye la tecnología requerida para unir los componentes y la información (por ejemplo, documentos, CD-ROM, vídeo) que se emplea para dar soporte a los componentes.

Como se muestra en la Figura 10.3, la visión global se consigue a través de *la ingeniería de requisitos.* Los requisitos generales del producto se obtienen del cliente. Estos requisitos comprenden necesidades de información y control, funcionalidad del producto y comportamiento, rendimiento general del producto, diseño, restricciones de la interfaz y otras necesidades especiales. Una vez que se conocen estos requisitos, la misión del análisis del sistema es asignar funcionalidad y comportamiento a cada uno de los cuatro componentes mencionados anteriormente.

Una vez que se ha hecho la asignación, comienza la *ingeniería de componentes del sistema*. La ingeniería de componentes del sistema es, de hecho, un conjunto de actividades concurrentes que se dirigen separadamente a cada uno de los componentes del sistema: la ingeniería del software, ingeniería hardware, ingeniería humana e ingeniería de bases de datos. Cada una de estas disciplinas de ingeniería toma una vista de dominio específica, pero es importante resaltar que las disciplinas de ingeniería deben establecer y mantener una comunicación activa entre ellas.

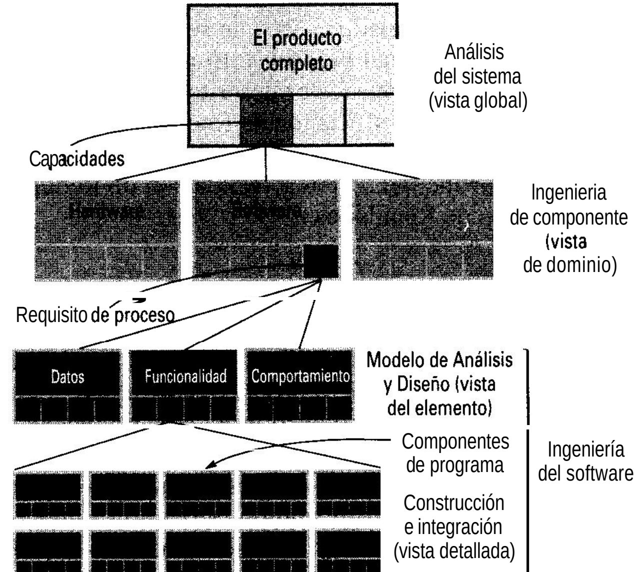


FIGURA 10.3. La jerarquía de la ingeniería de productos.

Parte del papel del análisis de sistemas es establecer los mecanismos de interfaz que permitirán que esto suceda.

La visión de elemento para la ingeniería de producto es la disciplina de ingeniería aplicada a la asignación de componentes. Para la ingeniería del software, esto significa *actividades de modelado del análisis y diseño* (cubierto en detalle en posteriores capítulos) y *actividades de construcción e integración* que comprenden generación de código, pruebas y actividades de soporte. El modelado de la fase de análisis asigna requisitos a las representaciones de datos, funciones y comportamiento. El diseño convierte el modelo de análisis en diseños de datos, arquitectónicos, de interfaz y a nivel de componentes del software.

10.5 INGENIERÍA DE REQUISITOS

La consecuencia del proceso de ingeniería de sistemas es la especificación de un sistema o producto basado en computadora que se describe genéricamente, en diferentes niveles en la Figura 10.1. Pero el desafío de la ingeniería del sistema (y de los ingenieros del software) es importante: ¿Cómo podemos asegurar que hemos especificado un sistema que recoge las necesidades del cliente y satisface sus expectativas? No hay una respuesta segura a esta difícil pregunta, pero un sólido proceso de ingeniería de requisitos es la mejor solución de que disponemos actualmente.


Cita:

La parte más dura en la construcción de un sistema software es decidir cómo construirlo... Ninguna parte del trabajo mutila el resultado del sistema si está hecho mal. Ninguna parte es más dificultosa para rectificarlo después.

Fred Brooks

La ingeniería de requisitos facilita el mecanismo apropiado para comprender lo que quiere el cliente, ana-

[4] Debemos hacer notar que la terminología (adaptada por [MAR90]) utilizada en la Figura 10.2 está asociada con la ingeniería de la información, la predecesora del moderno IPN. Sin embargo, el área central implicada en cada actividad señalada es dirigida por todos aquellos que consideran el tema.

lizando necesidades, confirmando su viabilidad, negociando una solución razonable, especificando la solución sin ambigüedad, validando la especificación y gestionando los requisitos para que se transformen en un sistema operacional [THA97]. El proceso de ingeniería de requisitos puede ser descrito en **5** pasos distintos [SOM97]: Identificación de Requisitos, Análisis de Requisitos y Negociación, Especificación de Requisitos, Modelizado del Sistema, Validación de Requisitos y Gestión de Requisitos.

10.5.1. Identificación de requisitos

En principio, parece bastante simple — preguntar al cliente, a los usuarios y a los que están involucrados en los objetivos del sistema o producto y sean expertos, investigar cómo los sistemas o productos se ajustan a las necesidades del negocio, y finalmente, cómo el sistema o producto va a ser utilizado en el día a día—. Esto que parece simple, es muy complicado.

Christel y Kang [CRI92] identifican una serie de problemas que nos ayudan a comprender por qué la obtención de requisitos es costosa.

- *problemas de alcance.* El límite del sistema está mal definido o los detalles técnicos innecesarios, que han sido aportados por los clientes/usuarios, pueden confundir más que clarificar los objetivos del sistema.
- *problemas de comprensión.* **Los** clientes/usuarios no están completamente seguros de lo que necesitan, tienen una pobre compresión de las capacidades **y** limitaciones de su entorno de computación, no existe un total entendimiento del dominio del problema, existen dificultades para comunicar las necesidades al ingeniero del sistema, la omisión de información por considerar que es «obvia», especificación de requisitos que están en conflicto con las necesidades de otros clientes/usuarios, o especificar requisitos ambiguos o poco estables.
- *Problemas de volatilidad.* Los requisitos cambian con el tiempo.



Un informe detallado bajo el titulo «Necesidades en la Obtención de Requisitos» puede ser descargado de **www.sei.cmu.edu/publications/documents/92.reports/92.tr.012.html**

Para ayudar a solucionar estos problemas, los ingenieros de sistemas deben aproximarse de una manera organizada a través de reuniones para definir requisitos.

Sommerville y Sawyer [SOM97] sugieren un conjunto de actuaciones para la obtención de requisitos, que están descritos en las tareas siguientes:

- valorar el impacto en el negocio **y** la viabilidad técnica del sistema propuesto;

¿Por qué es tan difícil obtener un planteamiento claro de lo que necesita el cliente?

- identificar las personas que ayudarán a especificar requisitos y contrastar su papel en la organización;
- definir el entorno técnico (arquitectura de computación, sistema operativo, necesidades de telecomunicaciones) en el sistema o producto a desarrollar e integrar;
- identificar «restricciones de dominio» (características específicas del entorno de negocio en el dominio de la aplicación) que limiten la funcionalidad y rendimientos del sistema o producto a construir;
- definir uno o más métodos de obtención de requisitos (entrevistas, grupos de trabajo, equipos de discusión);
- solicitar la participación de muchas personas para que los requisitos se definan desde diferentes puntos de vista, asegurarse de identificar lo fundamental de cada requisito registrado;
- identificar requisitos ambiguos como candidatos para el prototipado, **y**
- crear escenarios de uso (ver Capítulo **11**) para ayudar a los clientes/usuarios a identificar mejor los requisitos fundamentales.



*Asegúrate de haber **valorado** los características generales antes de especificar el esfuerzo y plazo para obtener los requisitos de detalle.*

El resultado alcanzado como consecuencia de la identificación de requisitos variará dependiendo del tamaño del sistema o producto a construir. Para grandes sistemas, el producto obtenido debe incluir:

- una relación de necesidades y características;
- un informe conciso del alcance del sistema o producto;
- una lista de clientes, usuarios y otros intervinientes que deben participar en la actividad de obtención de requisitos;
- una descripción del entorno técnico del sistema;
- una relación de requisitos (perfectamente agrupados por funcionalidad) y las restricciones del dominio aplicables a cada uno;
- un conjunto de escenarios que permiten profundizar en el uso del sistema o producto bajo diferentes condiciones operativas, y
- cualquier prototipo desarrollado para definir mejor los requisitos.

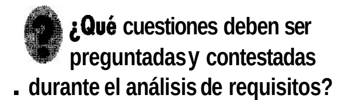
Referencia cruzada

los métodos de obtención de requisitos son presentados en el Capítulo 11.

Cada uno de los productos obtenidos debe ser revisado por las personas que hayan participado en la obtención de sus requisitos.

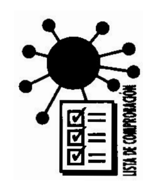
10.5.2. Análisis y negociación de requisitos

Una vez recopilados los requisitos, el producto obtenido configura la base del *análisis de requisitos.* Los requisitos se agrupan por categorías y se organizan en subconjuntos, se estudia cada requisito en relación con el resto, se examinan los requisitos en su consistencia, completitud y ambigüedad, y se clasifican en base a las necesidades de los clientes/usuarios.



Al iniciarse la actividad del análisis de requisitos se plantean las siguientes cuestiones:

- ¿Cada requisito es consistente con los objetivos generales del sistema/producto?
- ¿Tienen todos los requisitos especificados el nivel adecuado de abstracción? Es decir, ¿algunos requisitos tienen un nivel de detalle técnico inapropiado en esta etapa?
- ¿El requisito es necesario o representa una característica añadida que puede no ser esencial a la finalidad del sistema?
- ¿Cada requisito está delimitado y sin ambigüedad?
- Cada requisito tiene procedencia. Es decir, ¿existe un origen (general o específico) conocido para cada requisito?
- ¿Existen requisitos incompatibles con otros requisitos?
- ¿Es posible lograr cada requisito en el entorno técnico donde se integrará el sistema o producto?
- ¿Se puede probar el requisito una vez implementado?



Análisis de Requisitos

Es corriente en clientes y usuarios solicitar más de lo que puede realizarse, consumiendo recursos de negocio limitados. También es relativamente común en clientes y usuarios el proponer requisitos contradictorios, argumentando que su versión es «esencial por necesidades especiales».



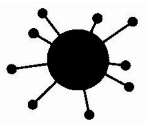
Si los diferentes clientes/usuarios no pueden facilitar los requisitos, el riesgo de errar es muy alto. Proceda con extrema precaución.

El ingeniero del sistema debe resolver estos conflictos a través de un proceso de negociación. Los clientes, usuarios y el resto de intervinientes deberán clasificar sus requisitos y discutir los posibles conflictos según su prioridad. Los riesgos asociados con cada requisito serán identificados y analizados (ver Capítulo 6). Se efectúan «estimaciones» del esfuerzo de desarrollo que se utilizan para valorar el impacto de cada requisito en el coste del proyecto y en el plazo de entrega. Utilizando un procedimiento iterativo, se irán eliminando requisitos, se irán combinando y/o modificando para conseguir satisfacer los objetivos planteados.

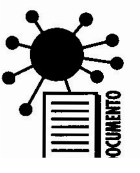
10.5.3. Especificación de requisitos

En el contexto de un sistema basado en computadoras (y software), el término *especificación* significa distintas cosas para diferentes personas. Una especificación puede ser un documento escrito, un modelo gráfico, un modelo matemático formal, una colección de escenarios de uso, un prototipo o una combinación de lo anteriormente citado.

Algunos sugieren que debe desarrollarse una «plantilla estándar» [SOM97] y usarse en la especificación del sistema, argumentando que así se conseguirían requisitos que sean presentados de una forma más consistente y más comprensible. No obstante, en muchas ocasiones es necesario buscar la flexibilidad cuando una especificación va a ser desarrollada. Para grandes sistemas, un documento escrito, combinado con descripciones en lenguajes natural y modelos gráficos puede ser la mejor alternativa. En cualquier caso, los escenarios a utilizar pueden ser tanto los requeridos para productos de tamaño pequeño o los de sistemas que residan en entornos técnicos bien conocidos.


Técnicas de Negociación

La *Especificación del Sistema* es el producto final sobre los requisitos del sistema obtenido por el ingeniero. Sirve como fundamento para la ingeniería del hardware, ingeniería del software, la ingeniería de bases de datos y la ingeniería humana. Describe la función y características de un sistema de computación y las restricciones que gobiernan su desarrollo. La especificación delimita cada elemento del sistema. La *Especificación del Sistema* describe la información (datos y control) que entra y sale del sistema.



Especificación del sistema

10.5.4. Modelado del sistema

Considere por un momento que a usted se le ha requerido para especificar los requisitos para la construcción de una cocina. Se conocen las dimensiones del lugar, la localización de las puertas y ventanas y el espacio de pared disponible. Debes especificar todos los armarios y electrodomésticos e indicar dónde colocarlos. ¿Será una especificación válida?

La respuesta es obvia. Para especificar completamente lo que vamos a desarrollar, necesitamos un modelo de la cocina con toda su información, esto es, un anteproyecto o una representación en tres dimensiones que muestre las posiciones de los armarios y electrodomésticos, y sus relaciones. Con el modelo será relativamente fácil asegurar la eficiencia del trabajo (un requisito de todas las cocinas), la estética «visual» de la sala (es un requisito personal, aunque muy importante).

Se construyen modelos del sistema por la misma razón que desarrollamos para una cocina un anteproyecto o una representación en 3D. Es importante evaluar los componentes del sistema y sus relaciones entre sí; determinar cómo están reflejados los requisitos, y valorar como se ha concebido la «estética» en el sistema. Se profundiza en el modelado del sistema en la Sección **10.6.**

10.5.5. Validación de requisitos

El resultado del trabajo realizado es una consecuencia de la ingeniería de requisitos (especificación del sistema e información relacionada) y es evaluada su calidad en la fase de validación. La *validación de requisitos* examina las especificaciones para asegurar que todos los requisitos del sistema han sido establecidos sin ambigüedad, sin inconsistencias, sin omisiones, que los errores detectados hayan sido corregidos, y que el resultado del trabajo se ajusta a los estándares establecidos para el proceso, el proyecto y el producto.

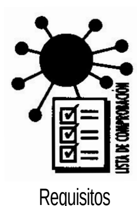


Un punto fundamental durante la validación de los requisitos es la consistencia. Utilice el modelo del sistema para asegurar que los requisitos han sido consistentemente establecidos.

El primer mecanismo para la validación de los requisitos es la revisión técnica formal (Capítulo **8).** El equipo de revisión incluye ingenieros del sistema, clientes, usuarios, y otros intervinientes que examinan la especificación del sistema[5] buscando errores en el contenido o en la interpretación, áreas donde se necesitan aclaraciones, información incompleta, inconsistencias (es un problema importante), requisitos contradictorios, o requisitos imposibles o inalcanzables.

Aunque la validación de requisitos puede guiarse de manera que se descubran errores, es útil chequear cada requisito con un cuestionario. Las siguientes cuestiones representan un pequeño subconjunto de las preguntas que pueden plantearse:

- ¿Está el requisito claramente definido? ¿Puede interpretarse mal?
- ¿Está identificado el origen del requisito (por ejemplo: persona, norma, documento)? ¿El planteamiento final del requisito ha sido contrastado con la fuente original?
- ¿El requisito está delimitado en términos cuantitativos?
- ¿Qué otros requisitos hacen referencia al requisito estudiado? ¿Están claramente identificados por medio de una matriz de referencias cruzadas o por cualquier otro mecanismo?
- ¿El requisito incumple alguna restricción definida?
- ¿El requisito es verificable? Si es así, ¿podemos efectuar pruebas (algunas veces llamadas criterios de validación) para verificar el requisito?
- ¿Se puede seguir el requisito en el modelo del sistema que hemos desarrollado?
- ¿Se puede localizar el requisito en el conjunto de objetivos del sistema/producto?
- ¿Está el requisito asociado con los rendimientos del sistema o con su comportamiento y han sido establecidas claramente sus características operacionales? ¿El requisito está implícitamente definido?



Requisitos

Las preguntas planteadas en la lista de comprobación ayudan a asegurar que el equipo de validación dispone de lo necesario para realizar una revisión completa de cada requisito.

10.5.6. Gestión de requisitos

En el capítulo anterior se advertía que los requisitos del sistema cambian y que el deseo de cambiar los requisitos persiste a lo largo de la vida del sistema. La *gestión de requisitos* es un conjunto de actividades que ayudan al equipo de trabajo a identificar, controlar y seguir los

[5] En realidad, muchas Revisiones Técnicas Formales son dirigidas a medida que se desarrolla la especificación del sistema. Es mejor para el equipo de revisión examinar pequeñas partes de la especificación, de forma que se pueda centrar la atención en un aspecto específico de los requisitos.

requisitos y los cambios en cualquier momento. Muchas de estas actividades son idénticas a las técnicas de gestión de configuración del software que se referencian en el Capítulo 9.



Un artículo titulado ((Configurando el trabajo de gestión de requisitos para ti» contiene una guía pragmático:
stsc.hill.af.mil./crosstalk/1999/apr/davis.asp

Como en la Gestión de Configuración del Software (GCS), la gestión de requisitos comienza con la actividad de identificación. A cada requisito se le asigna un Único identificador, que puede tomar la forma:

<tipo de requisito><requisito n.° >

El tipo de requisito toma alguno de los siguientes valores: F = requisito funcional, D = requisito de datos, C = requisito de comportamiento, I = requisito de interfaz, y S = requisito de salida. De esta forma, un requisito identificado como F09 indica que se trata de un requisito funcional y que tiene asignado el número 9 dentro de los citados requisitos.



Muchas actividades de gestión de requisitos son tomadas de la GCS.

Una vez los requisitos han sido identificados, se desarrollarán un conjunto de matrices para su seguimiento. En la Figura **10.4** se muestra de forma esquemática este planteamiento. Cada matriz de seguimiento identifica los requisitos relacionados con uno o más aspectos del sistema o su entorno. Entre las posibles matrices de seguimiento citamos las siguientes:

Matriz de seguimiento de características. Muestra los requisitos identificados en relación a las características definidas por el cliente del sistema/producto.

Matriz de seguimiento de orígenes. Identifica el origen de cada requisito.

Matriz de seguimiento de dependencias. Indica cómo se relacionan los requisitos entre **sí.**

Matriz de Seguimiento de subsistemas. Vincula los requisitos a los subsistemas que los manejan.

Matriz de seguimiento de interfaces. Muestra como los requisitos están vinculados a las interfaces externas o internas del sistema.



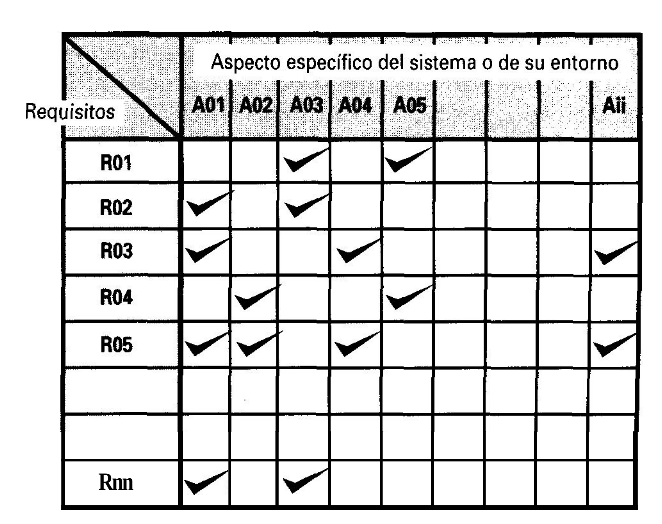
Cuando un sistema es grande y *complejo,* determinar las *«conexiones»* entre las *requisitos* puede ser una tarea desalentadora. Utilice *las tablas* de seguimiento para hacer el trabajo un poco *más fácil.*

En muchos casos, las matrices de seguimiento se incorporan como parte de un requisito de base de datos y se utiliza para buscar rápidamente los diferentes aspectos del sistema a construir afectados por el cambio de requisito.

10.6 MODELADO DEL SISTEMA

Todos los sistemas basados en computadora pueden modelarse como una transformación de la información empleando una arquitectura del tipo entrada-proceso-salida. Hatley y Pirbhai [HAT87] han extendido esta visión para incluir dos características adicionales del sistema: tratamiento de la interfaz de usuario y tratamiento del mantenimiento y autocomprobación. Aunque estas características adicionales no están presentes en todos los sistemas basados en computadora, son muy comunes y su especificación hace más robusto cualquier modelo del sistema.

Mediante la representación de entrada, proceso, salida, tratamiento de la interfaz de usuario y de autocomprobación, un ingeniero de sistemas puede crear un modelo de componentes de sistema que establezca el fundamento para análisis de requisitos posteriores y etapas de diseño en cada una de las disciplinas de ingeniería.



Requisitos	Aspecto específico del sistema o de su entorno								
	A01	A02	A03	A04	A05				Aii
R01			✓		✓				
R02	✓		✓						
R03	✓			✓					✓
R04		✓			✓				
R05	✓	✓		✓					✓
Rnn	✓		✓						

FIGURA 10.4. Matriz de seguimiento genérica.

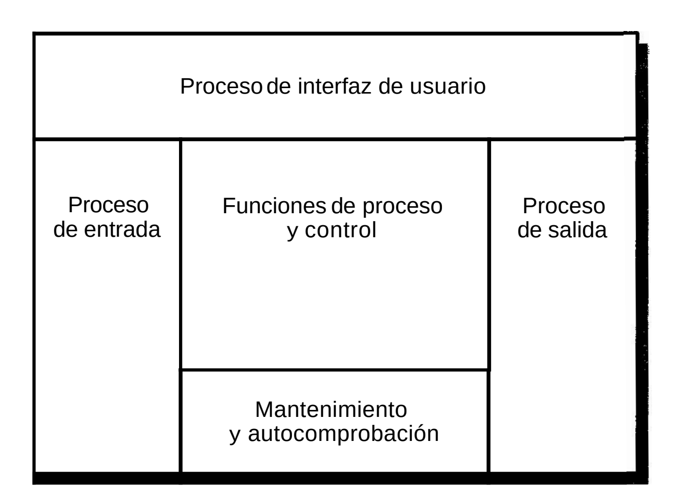


FIGURA 10.5. Plantilla del modelo del sistema [HAT87].

Para desarrollar el modelo de sistema, se emplea un *esquema del modelado del sistema* [HAT87]. El ingeniero de sistemas asigna elementos a cada una de las cinco regiones de tratamiento del esquema: (1) interfaz de usuario, (2) entrada, *(3)* tratamiento y control del sistema, (4) salida y *(5)* mantenimiento y autocomprobación. En la Figura **10.5** se muestra el formato del esquema de arquitectura.

Referencia cruzada

Otros métodos de modelización del sistema toman una visión orientada a objetos. El enfoque UML puede ser aplicado a estos sistemas, planteándose en los Capítulos 21 y 22.

Como casi todas las técnicas de modelado usadas en la ingeniería del software y de sistemas, el esquema del modelado del sistema permite al analista crear una jerarquía en detalle. En la parte alta de la jerarquía reside *el diagrama de contexto del sistema (DCS).* El diagrama de contexto «establece el límite de información entre el sistema que se está implementando y el entorno en que va a operar» [HAT87]. Es decir, el DCS define todos los suministradores externos de información que emplea el sistema, todos los consumidores externos de información creados por el sistema y todas las entidades que se comunican a través de la interfaz o realizan mantenimiento y autocomprobación.

Para ilustrar el empleo del DCS, considere una versión ampliada del *sistema de clasificación de cinta transportadora (SCCT)* estudiado anteriormente en el Capítulo 5. Al ingeniero del sistema se le presenta la siguiente declaración (algo confusa) de objetivos para el SCCT:

El sistema SCCT debe desarrollarse de manera que las cajas que se mueven a lo largo de la cinta transportadora sean identificadas y ordenadas en uno de los seis contenedores al final de la cinta. Las cajas pasarán a través de una estación clasificadora donde se identificarán. Basándose en un número de identificación impreso en un lateral de la caja (se proporciona un código de barras equivalente), las cajas se mandarán a los contenedores apropiados. Las cajas pasan aleatoriamente y están igualmente espaciadas. La línea se mueve lentamente.

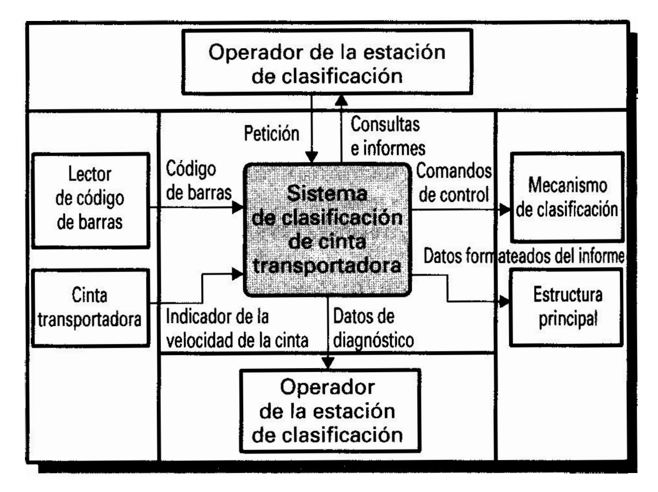


FIGURA 10.6. Diagrama de contexto de Arquitectura del SCCT (ampliado).

Para este ejemplo, la versión ampliada utiliza un PC en la estación clasificadora. El PC ejecuta todo el software del SCCT; interacciona con el lector de código de barras para leer parte de los números de cada caja; interacciona con la cinta transportadora vigilando los equipos que controlan velocidad en dicha cinta; almacena todos los números de pieza clasificados; interacciona con el operador de la estación clasificadora para producir una variedad de informes y diagnósticos; envía señales de control al hardware separador para clasificar las cajas; y se comunica con la estructura central de la automatización de la fábrica. En la Figura **10.6** se muestra el DCS para SCCT (ampliado).



El DCS permite una visión «global» del sistema que debes construir. los detalles que necesites no deben especificarse en este nivel. Refina jerárquicamente el DCS para elaborar el sistema.

Cada caja de la Figura **10.6** representa una *entidad externa,* es decir, un suministrador o consumidor de información del sistema. Por ejemplo, el lector del código de barras produce información que es introducida en el sistema SCCT. El símbolo para representar todo el sistema (o a niveles más bajos, subsistemas principales) es un rectángulo con las esquinas redondeadas. De ahí que SCCT se represente en la región de proceso y control en el centro del DCS. Las flechas etiquetadas mostradas en el DCS representan información (datos y control) que va desde el entorno exterior al sistema SCCT. La entidad externa lector del código de barras produce una entrada de información etiquetada como **código de barras.** En esencia, el DCS sitúa a cualquier sistema en el contexto de su entorno exterior.

Interfaz el operador
Peticiones de operador
Subsistema de interfaz con el operador
Respuestas, informes y visualizaciones del SCCT
Subsistema lector de código de barras
Código de barras
Subsistema del sensor de adquisición de datos
Entrada de tacómetro de impulsos
Interfaz de adquisición de datos
Proceso y control del SCCT
Subsistema decodificador del código de barras
Número de piezas
Datos del código de barras
Velocidad de la cinta
Subsistema de acceso a la base de datos
Petición de informe
Datos de
Subsistema de control de maniobras
Posición del contenedor
Clave
Subsistema de configuración de informes
Clasificación de registros
Estado BCR
Estado del sensor
Subsistema de diagnósticos
Estado de maniobra
Estado de comunicaciones
Estado del lector de código de barras
Interfaz de diagnóstico
calización temporal
Controlador de maniobras
Órdenes de maniobra
Informes de SCCT
Controlador de las comunicaciones con la computadora central
Configuración de los datos del informe
Interfaz de salida

FIGURA 10.7. Diagrama de flujo de arquitectura para el SCCT (ampliado).

Referencia cruzada

El DCS es un precursor del diagrama de flujo de datos, que se estudia en el Capítulo 12.

El ingeniero de sistemas refina el diagrama de contexto de arquitectura estudiando con más detalle el rectángulo sombreado de la Figura 10.6. Se identifican los subsistemas principales que permiten funcionar al sistema clasificador de cinta transportadora dentro del contexto definido por el DCS. En la Figura 10.7 los subsistemas principales se definen en un *diagrama de flujo del sistema (DFS)* que se obtiene del DCS. El flujo de información a través de las regiones del DCS se usa para guiar al ingeniero de sistemas en el desarrollo de DFS (un «esquema» más detallado del SCCT). El diagrama de flujo de la arquitectura muestra los subsistemas principales y el flujo de las líneas de información importantes (datos y control). Además, el esquema del sistema divide el proceso del subsistema en cada una de las cinco regiones de proceso estudiadas anteriormente. En este punto, cada uno de los subsistemas puede contener uno o más elementos del sistema (por ejemplo, hardware, software, personas) tal y como los ha asignado el ingeniero de sistemas.

El diagrama inicial de flujo del sistema (DFS) se convierte en el nudo superior de una jerarquía de DFS. Cada rectángulo redondeado del DFS original puede expandirse en otra plantilla de arquitectura dedicada exclusivamente a ella. Este proceso se ilustra esquemáticamente en la Figura 10.8. Cada uno de los DFS del sistema puede usarse como punto de partida de subsiguientes fases de ingeniería para el subsistema que se describe.



Referencia Web

Para concretar el método de Hatley-Pirbhai se puede acudir a **www.hasys.com/papers/hp_description.html**

Se pueden especificar (delimitar) los subsistemas y la información que fluyen entre ellos para los subsiguientes trabajos de ingeniería. Una descripción narrativa de cada subsistema y una definición de todos los datos que fluyen entre los subsistemas son elementos importantes de la *Especificación del Sistema.*

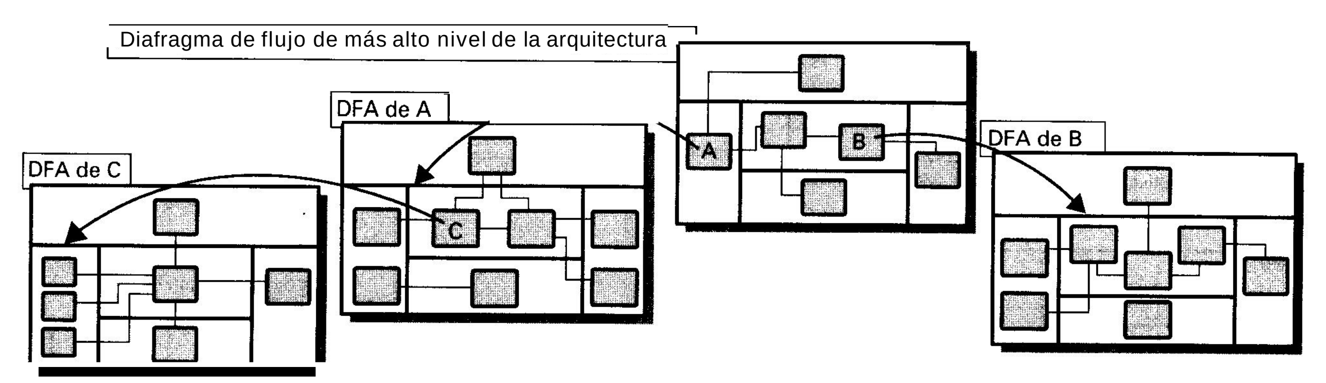


FIGURA 10.8. Construcción de una jerarquía **DFA.**

RESUMEN

Un sistema de alta tecnología comprende varios componentes: hardware, software, personas, bases de datos, documentación y procedimientos. La ingeniería de sistemas ayuda a traducir las necesidades del cliente en un modelo.de sistema que utiliza uno o más de estos componentes.

La ingeniería de sistemas comienza tomando una «visión global». Se analiza el dominio de negocio o producto para establecer todos los requisitos básicos. El enfoque se estrecha entonces a una «visión de dominio», donde cada uno de los elementos del sistema se analiza individualmente. Cada elemento es asignado a uno o más componentes de ingeniería que son estudiados por la disciplina de ingeniería correspondiente.

La ingeniería de procesos de negocio es un enfoque de la ingeniería de sistemas que se usa para definir arquitecturas que permitan a un negocio utlizar la información eficazmente. La intención de la ingeniería de procesos de negocio es crear minuciosas arquitecturas de datos, una arquitectura de aplicación y una infraestructura tecnológica que satisfaga las necesidades de la estrategía de negocio y los objetivos de cada área de negocio. La ingeniería de procesos de negocio comprende una *planificación de la estrategia de la información (PEI)*, un *análisis del área de negocio (AAN)* y un análisis específico de aplicación que de hecho forman parte de la ingeniería del software.

La ingeniería de productos es un enfoque de la ingeniería de sistemas que empieza con el análisis del sistema. El ingeniero de sistemas identifica las necesidades del cliente, determina la viabilidad económica y técnica, y asigna funciones y rendimientos al software, hardware, personas y bases de datos; los componente claves de la ingeniería.

La ingeniería del sistema demanda una intensa comunicación entre el cliente y el ingeniero del sistema. Esto se realiza a través de un conjunto de actividades bajo la denominación de ingeniería de requisitos –identificación, análisis y negociación, especificación, modelización, validación y gestión–.

Una vez que los requisitos hayan sido aislados, el modelado del sistema puede ser realizado, y las representaciones de los subsistemas principales pueden ser desarrolladas. La tarea de la ingeniería del sistema finaliza con la elaboración de una *Especificación del Sistema* –un documento que sirve de base para las tareas de ingeniería que se realizarán posteriormente–.

REFERENCIAS

[CRI92] Chnstel, M.G., y K.C. Kang, «Issuesin Requirements Elicitation», Software Engineering Institute, CMU/SEI-92-TR-12 7, September 1992.

[GRA69] Graham, R.M., en *Proceedings 1969 NATO Conference* ***on*** *Sofiare Engineering,* 1969.

[GUT99] Guttman, M., «Architectural Requirements for a Changing Business World», *Research Briefsfrom Cutter Consortium (an online service),* June 1, 1999.

[HAR93] Hares, J.S., *Information Engineeringfor the Advanced Practitioner,* Wiley, 1993, pp. 12-13.

[HAT87] Hatley, D.J., e I.A. Pirbhai, *Strategiesfor Real-Time System Specification*, Dorset House, 1987.

[MAR90] Martin, J., *Information Engineering: Book II – Planning and Analysis*, Prentice Hall, 1990.

[MOT92] Motamarri, S., «Systems Modelling and Description», *Software Engineering Notes,* vol. 17, n.° 2, Abril 1992, pp. 57-63.

[SOM97] Somerville, I., y P. Sawyer, *Requirements Engineering*, Wiley, 1997

[SPE93] Spewak, S., *Enterprise Architecture Planning*, QED Publishing, 1993.

[THA97] Thayer, R.H., y M. Dorfman, *Software Requirements Engineering,* 2.ª ed., IEEE Computer Society Press, 1997.

PROBLEMAS Y PUNTOS A CONSIDERAR

10.1. Encuentre tantos sinónimos como pueda de la palabra «sistema». ¡Buena suerte!

10.2. Construya una descripción en estructura jerárquica para un sistema, producto o servicio que le sea familiar. El planteamiento abarcará los elementos fundamentales del sistema (hardware, software, etc.) de una rama concreta de dicha estructura.

10.3. Seleccione un gran sistema o producto con el que esté familiarizado. Defina el conjunto de dominios que describen la visión global de él. Describa el conjunto de elementos que compongan uno o dos dominios. Para un elemento, identifique los componentes técnicos con los que hay que hacer ingeniería.

10.4. Seleccione un gran sistema o producto con el que esté familiarizado. Establezca los supuestos, simplificaciones, limitaciones, restricciones y preferencias que deberían haberse hecho para construir un modelo de sistema eficaz y realizable.

10.5. La ingeniería de proceso de negocio requiere definir datos, arquitectura.de aplicaciones, además de una infraestructura tecnológica. Describa cada uno de estos términos mediante un ejemplo.

10.6. La planificación de la estrategía de la información empieza por la definición de objetivos. Ponga ejemplos de cado uno de los dominios de negocio.

10.7. Un ingeniero de sistemas puede venir de una de tres procedencias: el desarrollo de sistemas, el cliente o una tercera organización. Discuta los pros y contras de cada procedencia. Descnba a un ingeniero de sistemas «ideal».

10.8. Su profesor les repartirá una descripción de alto nivel de un sistema o producto.

a. Desarrolle un conjunto de cuestiones que debería preguntar como ingeniero de sistemas.

b. Proponga al menos dos asignaciones diferentes para el sistema basándose en las respuestas de **su** profesor a sus preguntas.

c. En clase, compare sus respuestas con las de sus compañeros.

10.9. Desarrolle una lista de comprobación para los atributos a considerar cuando hay que evaluar la «viabilidad» de un sistema o producto. Estudie la interacción entre los atributos e intente proporcionar un método para puntuar cada uno de manera que se obtenga una «puntuación de viabilidad».

10.10. Investigue las técnicas de contabilidad que se emplean para un análisis detallado de coste/beneficio de un sistema que requiera algo de fabricación y montaje de hardware. Intente escribir un libro de directrices que pueda aplicar un gestor técnico.

10.11. Desarrolle el diagrama de contexto DCS y el resto de diagramas de flujo de un sistema a su elección (o definido por su profesor).

10.12. Escriba una descripción de un módulo del sistema que pudiera estar contenido en la especificación de diagrama de arquitectura de uno o más de los subsistemas definidos en los DFA desarrollados para el problema 10.11.

10.13. Investigue documentación sobre herramientas CASE y escriba un breve documento describiendo cómo trabajan las herramientas de modelado y simulación. Alternativa: Reúna información de dos o más vendedores de CASE que suministren herramientas de modelado y simulación y valore las similitudes y las diferencias.

10.14. Basándose en la documentación que le proporcione su profesor, desarrolle una pequeña *Especificación de Sistema* para uno de los siguientes sistemas:

a. Un sistema de edición de vídeo digital no lineal

b. Un escáner digital para ordenador personal

c. Un sistema de correo electrónico

d. Un sistema para apuntarse a la universidad

e. Un proveedor de acceso a internet

f. Un sistema interactivo de reserva de hoteles

g. Un sistema de interés local

Asegúrese de crear los modelos de arquitectura descritos en la Sección 10.6

10.15. ¿Existen características de un sistema que no se puedan establecer durante las actividades de ingeniería de sistemas? Describa las características, si existen, y explique por qué se debe retrasar su consideración a fases posteriores de la ingeniería.

10.16. ¿Existen situaciones en las que se pueda abreviar o eliminar completamente la especificación formal del sistema? Explíquelo.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Se han publicado relativamente pocos libros sobre ingeniería de sistemas en los últimos años. Entre ellos destacamos:

Blanchard, B.S., *System Engineering Management,* 2.ª ed., Wiley, 1997.

Rechtin, E., y M.W. Maier, *The Art of Systems Architecting*, CRC Press, 1996.

Weiss, D., et al., *Software Product-Line Engineering,* Addison-Wesley, 1999.

Libros como los de Armstrong y Sage (*Intoduction to Systems Engineering,* Wiley, 1997), Martin (*Systems Engineering Guidehook,* CRC Press, 1996), Wymore (*Model-Based Systems Engineering,* CRC Press, 1993), Lacy (*System Engineering Management,* McGraw-Hill, 1992), Aslaksen y Belcher (*Systems Engineering,* Prentice-Hall, 1992), Athey (*Systematic Systems Approach,* Prentice-Hall, 1982), y Blanchard y Fabrycky (*Systems Engineering and Analysis,* Prentice-Hall, 1981) presentan el proceso de la ingeniería del sistema (con un énfasis diferente de la ingeniería) y ofrecen una valiosa guía.

En los Últimos años, los textos de ingeniería de la información han sido reemplazados por libros que se centran en

la ingeniería de proceso de negocio. Scheer ***(Business Process Engineering: Reference Modelsfor Industrial Enterprises,*** Springer-Verlag, 1998) describe métodos de modelado de procesos de negocio para empresas con amplios sistemas de información. Lozinsky ***(Enterprise-Wide Software Solutions: Integration Strategies and Practices,*** Addison-Wesley, 1998) plantea el uso de paquetes software como una solución que permita a una compañía migrar de los sistemas heredados a procesos de negocio modernos. Martin ***(Information Engineering,*** 3 volúmenes, Prentice-Hall, 1989, 1990, 1991) presenta un planteamiento comprensivo sobre los tópicos de la ingeniería de la información. Libros como los de Hares [HAR93], Spewak [SPE93] y Flynn y Fragoso-Diaz (***Information Modeling: An International Perspective,*** Prentice-Hall, 1996) tratan este tema con detalle.

Davis y Yen ***(The Information** system **Consultant's Handbook: Systems Analysis and Design,*** CRC Press, 1998) presentan una cobertura enciclopédica de los resultados del análisis y diseño de sistemas en el dominio de los sitemas de información. Un volumen más reciente de **los** mismos autores ***(Standards, Guidelines and Examples: System and Software Requirements Engineering,*** IEEE Computer Society Press, 1990) presenta un planteamiento de estándares y directrices para el trabajo de análisis.

Para aquellos lectores involucrados activamente en el trabajo de sistemas o interesados en un tratamiento más sofisticado sobre el tema, se propone los libros de Gerald Weinberg's ***(An Introduction to General System Thinking,*** Wiley- Interscience, 1976, y *On* ***the Design*** *of* ***Stable Systems,*** Wilwy-Interscience, 1979) permiten una excelente discusión sobre el «pensamiento general de sistemas» que implícitamentelleva a un acercamiento general al análisis y diseño de sistemas. Libros más recientes son los de Weinberg ***(General Principies*** *of* ***Systems Design, Dorset House,*** 1998, y ***Rethinking** systems Analysis* ***and Design,*** Dorset House, 1988) continúan en la tradición de **su** más avanzado trabajo.

Una amplia variación de fuentes de información sobre ingeniería de sistemas y elementos relacionados están disponibles en intemet. Una lista actualizada de referencias a páginas web sobre ingeniería de sistemas, ingeniería de la información, ingeniería de procesos de negocio e ingeniería de productos pueden ser encontradas en **http://www.pressman5.com**

CAPÍTULO

11 CONCEPTOS Y PRINCIPIOS DEL ANÁLISIS

LA ingeniería de requisitos del software es un proceso de descubrimiento, refinamiento, modelado y especificación. Se refinan en detalle los requisitos del sistema y el papel asignado al software —inicialmente asignado por el ingeniero del sistema—. Se crean modelos de los requisitos de datos, flujo de información y control, y del comportamiento operativo. Se analizan soluciones alternativas y el modelo completo del análisis es creado. Donald Reifer [REI94] describe el proceso de ingeniería de requisitos del software en el siguiente párrafo:

> La ingeniería de requisitos es el uso sistemático de procedimientos, técnicas, lenguajes y herramientas para obtener con un coste reducido el análisis, documentación, evolución continua de las necesidades del usuario y la especificación del comportamiento externo de un sistema que satisfaga las necesidades del usuario. Tenga en cuenta que todas las disciplinas de la ingeniería son semejantes, la ingeniería de requisitos no se guía por conductas esporádicas, aleatorias o por modas pasajeras, si no que se debe basar en el uso sistemático de aproximaciones contrastadas.

Tanto el desarrollador como el cliente tienen un papel activo en la ingeniería de requisitos del software —un conjunto de actividades que son denominadas *análisis*—. El cliente intenta replantear un sistema confuso, a nivel de descripción de datos, funciones y comportamiento, en detalles concretos. El desarrollador actúa como un interrogador, como consultor, como persona que resuelve problemas y como negociador.

VISTAZO RÁPIDO

¿Qué es? El papel global del software en un gran sistema es identificado durante la ingeniería del sistema (Capítulo 10). De cualquier manera, es necesario considerar una visión lo más profunda posible del papel del software —para comprender los requisitos específicos que deben ser considerados en la construcción de un software de alta calidad—. Este es el trabajo del análisis de requisitos del software. Para realizar el trabajo adecuadamente, se deben seguir un conjunto de conceptos y principios fundamentales.

¿Quién lo hace? Generalmente, el ingeniero del software es quién realiza el análisis de requisitos. En cualquier caso, para aplicaciones de negocio complejas, un «analista de sistemas» —formado en los aspectos del negocio del dominio de la aplicación— puede realizar esta tarea.

¿Por qué es importante? Si no analizas, es muy probable que construyas una solución software muy elegante que resuelva incorrectamente el problema. El resultado es tiempo y dinero perdido, frustración personal y clientes descontentos.

¿Cuáles son los pasos? Los requisitos de datos, funciones y comportamiento son identificados por la obtención de información facilitada por el cliente. Los requisitos son refinados y analizados para verificar su claridad, completitud y consistencia. La especificación se incorpora al modelo del software y es validada tanto por el ingeniero del software como por los clientes/usuarios.

¿Cuál es el producto obtenido? Una representación efectiva del software debe ser producida como una consecuencia del análisis de requisitos. Tanto los requisitos del sistema como los requisitos del software pueden ser representados utilizando un prototipo, una especificación o un modelo simbólico.

¿Cómo puedo estar seguro de que lo he hecho correctamente? El resultado obtenido del análisis de requisitos del software debe ser revisado para conseguir: claridad, completitud y consistencia.

El análisis y la especificación de los requisitos puede parecer una tarea relativamente sencilla, pero las apariencias engañan. El contenido de comunicación es muy denso. Abundan las ocasiones para las malas interpretaciones o falta de información. Es muy probable que haya ambigüedad. El dilema al que se enfrenta el ingeniero del software puede entenderse muy bien repitiendo la famosa frase de un cliente anónimo: «Sé que cree que entendió lo que piensa que dije, pero no estoy seguro de que se dé cuenta de que lo que escuchó no es lo que yo quise decir.»

11.1 ANÁLISIS DE REQUISITOS

El *análisis de los requisitos* es una tarea de ingeniería del software que cubre el hueco entre la definición del software a nivel sistema y el diseño del software (Fig. 11.1). El análisis de requisitos permite al ingeniero de sistemas especificar las características operacionales del software (función, datos y rendimientos), indica la interfaz del software con *otros* elementos del sistema **y** establece las restricciones que debe cumplir el software.

Cita:
Gastamos mucho tiempo —la mayor parte del tiempo del proyecto— no implementando o probando, sino intentando decidir qué construir.
Brian Lawrence

El análisis de requisitos del software puede dividirse en cinco áreas de esfuerzo: (1) reconocimiento del problema, (2) evaluación y síntesis, (3) modelado, (4) especificación y *(5)* revisión. Inicialmente, el analista estudia la *Especificación del Sistema* (si existe alguna) y el *Plan del Proyecto de Software.* Es importante entender el software en el contexto de un sistema y revisar el ámbito del software que se empleó para generar las estimaciones de la planificación. **A** continuación, se debe establecer la comunicación para el análisis de manera que nos garantice un correcto reconocimiento del problema. El objetivo del analista es el reconocimiento de los elementos básicos del problema tal y como los percibe el cliente/usuario.

La evaluación del problema **y** la síntesis de la solución es la siguiente área principal de esfuerzo en el análisis. El analista debe definir todos los objetos de datos observables externamente, evaluar el flujo y contenido de la información, definir y elaborar todas las funciones del software, entender el comportamiento del software en el contexto de acontecimientos que afectan al sistema, establecer las características de la interfaz del sistema y descubrir restricciones adicionales del diseño. Cada una de estas tareas sirve para describir el problema de manera que se pueda sintetizar un enfoque o solución global.

Por ejemplo, un mayorista de recambios de automóviles necesita un sistema de control de inventario. El analista averigua que los problemas del sistema manual que se emplea actualmente son: (1) incapacidad de obtener rápidamente el estado de un componente; (2) dos o tres días de media para actualizar un archivo a base de tarjetas; (3) múltiples órdenes repetidas para el mismo vendedor debido a que no hay manera de asociar a los vendedores con los componentes, etc. Una vez que se han identificado los problemas, el analista determina qué información va a producir el nuevo sistema y qué información se le proporcionará al sistema. Por ejemplo, el cliente desea un informe diario que indique qué piezas se han tomado del inventario y cuántas piezas similares quedan. El cliente indica que los encargados del inventario registrarán el número de identificación de cada pieza cuando salga del inventario.

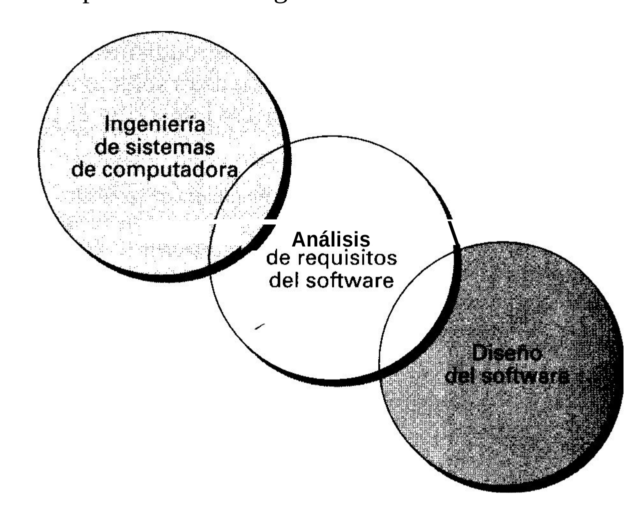


FIGURA 11.1. Análisis como puente entre la ingeniería y el diseño de software.



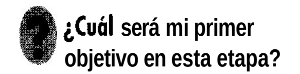
Cuento con hacer uno parte del diseño durante el análisis de requisitos y uno porte del análisis de requisitos durante el diseño.

Una vez evaluados los problemas actuales **y** la información deseada (entradas **y** salidas), el analista empieza a sintetizar una o más soluciones. Para empezar, se definen en detalle los datos, las funciones de tratamiento y el comportamiento del sistema. Una vez que se ha establecido esta información, se consideran las arquitecturas básicas para la implementación. Un enfoque cliente/servidor parecería apropiada, pero ¿está dentro del ámbito esbozado en el *Plan del Software?* Parece que sería necesario un sistema de gestión de bases de datos, pero, ¿está justificada la necesidad de asociación del usuario/cliente? El proceso de evaluación y síntesis continúa hasta que ambos, el analista y el cliente, se sienten seguros de que se puede especificar adecuadamente el software para posteriores fases de desarrollo.

A lo largo de la evaluación **y** síntesis de la solución, el enfoque primario del analista está en el «qué» **y** no en el «cómo». ¿Qué datos produce y consume el sistema, qué funciones debe realizar el sistema, qué interfaces se definen **y** qué restricciones son aplicables?[1].

[1] Davis [DAV93] argumenta que los términos «que» y «cómo» son demasiado vagos. Puede leer su libro si desea encontrar un interesante debate sobre el tema.

Durante la actividad de evaluación y síntesis de la solución, el analista crea modelos del sistema en un esfuerzo de entender mejor el flujo de datos y de control, el tratamiento funcional y el comportamiento operativo y el contenido de la información. El modelo sirve como fundamento para el diseño del software y como base para la creación de una especificación del software.



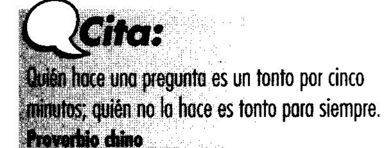
En el Capítulo 2, indicamos que puede no ser posible una especificación detallada en esta etapa. El cliente puede no estar seguro de lo que quiere. El desarrollador puede no estar seguro de que un enfoque específico consiga apropiadamente el funcionamiento y rendimiento adecuados. Por estas y otras muchas razones, se puede llevar a cabo un enfoque alternativo del análisis de requisitos, denominado *creación de prototipos o prototipado.* El prototipado se comenta más tarde en este capítulo.

11.2 IDENTIFICACIÓN DE REQUISITOS PARA EL SOFTWARE

Antes que los requisitos puedan ser analizados, modelados o especificados, deben ser recogidos a través de un proceso de obtención. Un cliente tiene un problema que pretende sea resuelto con una solución basada en computadora. Un desarrollador responde a la solicitud de ayuda del cliente. La comunicación ha empezado. Pero como ya hemos señalado, el camino de la comunicación al entendimiento está a menudo lleno de baches.

11.2.1. Inicio del proceso

La técnica de obtención de requisitos más usada es llevar a cabo una reunión o entrevista preliminar. La primera reunión entre el ingeniero del software (el analista) y el cliente puede compararse con la primera cita entre dos adolescentes. Nadie sabe qué decir o preguntar; los dos están preocupados de que lo que digan sea malentendido; ambos piensan qué pasará (los dos pueden tener expectativas radicalmente diferentes): los dos están deseando que aquello termine, pero, al mismo tiempo, ambos desean que la cita sea un éxito.



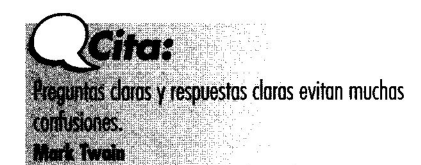
Sin embargo, hay que empezar la comunicación. Gause y Weinberg [GAU89] sugieren que el analista empiece preguntando cuestiones de contexto libre. Es decir, un conjunto de preguntas que llevarán a un entendimiento básico del problema, qué solución busca, la naturaleza de la solución que se desea y la efectividad del primer encuentro. El primer conjunto de cuestiones de contexto libre se enfoca sobre el cliente, los objetivos generales y los beneficios esperados. Por ejemplo, el analista podría preguntar:

- ¿Quién está detrás de la solicitud de este trabajo?
- ¿Quién utilizará la solución?
- ¿Cuál será el beneficio económico del éxito de una solución?
- ¿Hay alguna otra alternativa para la solución que necesita?

Estas preguntas ayudan a identificar todos los *participantes* que tienen un interés en el software a construir. Además, las preguntas identifican los beneficios medibles en una implementación correcta de posibles alternativas para un desarrollo normal del software.

El siguiente conjunto de preguntas permite al analista obtener un mejor entendimiento del problema y al cliente comentar **sus** opiniones sobre la solución:

- ¿Cómo caracterizaría una «buena» salida (resultado) generada para una buena solución?
- ¿A qué tipo de problema(s) va dirigida esta solución?
- ¿Puede mostrarme (o describirme) el entorno en que se utilizará la solución?
- ¿Hay aspectos o restricciones especiales del rendimiento que afecten a la manera de enfocar la solución?



El último conjunto de preguntas se concentra en la eficacia de la reunión. Gauge y Weinberg [GAU89] las denominan «meta-preguntas» y proponen la siguiente lista (abreviada):

- ¿Es usted la persona adecuada para responder a estas preguntas? ¿Sus respuestas son «oficiales»?
- ¿Estoy preguntando demasiado?
- ¿Hay alguien más que pueda proporcionar información adicional?
- ¿Hay algo más que debería preguntarle?



Si un sistema o producto va o servir poro muchos usuarios, los requisitos deberán ser obtenidos de un grupo representativo de usuarios. Si sólo uno persona define iodos los requisitos, el riesgo de aceptación será olio.

Estas preguntas (y otras) ayudarán a «romper el hielo» e iniciar la comunicación tan esencial para el éxito del análisis. Pero el formato de reunión tipo «pregunta

y respuesta» no es un enfoque que haya tenido mucho éxito. De hecho, esta sesión de preguntas y respuestas debería emplearse solamente en el primer encuentro y sustituirse después por una modalidad que combine elementos de resolución del problema, negociación y especificación. En la siguiente sección se presenta un enfoque a reuniones de este tipo.

11.2.2. Técnicas para facilitar las especificaciones de una aplicación

Los clientes y los ingenieros del software a menudo tienen una mentalidad inconsciente de «nosotros y ellos». En vez de trabajar como un equipo para identificar y refinar los requisitos, cada uno define por derecho su propio «territorio» y se comunica a través de una serie de memorandos, papeles de posiciones formales, documentos y sesiones de preguntas y respuestas. La historia ha demostrado que este método no funciona muy bien. Abundan los malentendidos, se omite información importante y nunca se establece una buena relación de trabajo.

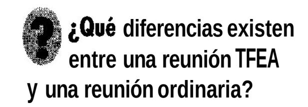


Una aproximación a FAST es llamada «Diseño común de aplicaciones» (JAD). Un detallado estudio de JAD puede encontrarse en
www.bee.net/bluebird/jaddoc.htm

Con estos problemas en mente es por lo que algunos investigadores independientes han desarrollado un enfoque orientado al equipo para la reunión de requisitos que se aplica durante las primeras fases del análisis y especificación. Denominadas *Técnicas para facilitar las especificaciones de la aplicación* ***(TFEA)***, este enfoque es partidario de la creación de un equipo conjunto de clientes y desarrolladores que trabajan juntos para identificar el problema, proponer soluciones, negociar diferentes enfoques y especificar un conjunto preliminar de requisitos de la solución [ZAH90]. Hoy en día, las TFEA son empleadas de forma general por los sistemas de información, pero la técnica ofrece un potencial de mejora en aplicaciones de todo tipo.

Se han propuesto muchos enfoques diferentes de TFEA[2]. Cada uno utiliza un escenario ligeramente diferente, pero todos aplican alguna variación en las siguientes directrices básicas:

- la reunión se celebra en un lugar neutral y acuden tanto los clientes como los desarrolladores.
- se establecen normas de preparación y de participación.
- se sugiere una agenda lo suficientemente formal como para cubrir todos los puntos importantes, pero lo suficientemente informal como para animar el libre flujo de ideas.
- un «coordinador» (que puede ser un cliente, un desarrollador o un tercero) que controle la reunión.
- se usa un «mecanismo de definición» (que puede ser hojas de trabajo, gráficos, carteles o pizarras).
- el objetivo es identificar el problema, proponer elementos de solución, negociar diferentes enfoques y especificar un conjunto preliminar de requisitos de la solución en una atmósfera que permita alcanzar el objetivo.



Para comprender mejor el flujo de acontecimientos tal y como ocurren en una reunión TFEA, presentamos un pequeño escenario que esboza la secuencia de acontecimientos que llevan a la reunión, los que ocurren en la reunión y los que la siguen.



En las reuniones iniciales entre el desarrollador y el cliente (Sección 11.2.1) se dan preguntas y respuestas básicas que ayudan a establecer el ámbito del problema y la percepción global de una solución. Fuera de estas reuniones iniciales, el cliente y el desarrollador escriben una «solicitud de producto» de una o dos páginas. Se selecciona lugar, fecha y hora de reunión TFEA y se elige un coordinador. Se invita a participar a representantes de ambas organizaciones; la del cliente y la de desarrollo. Se distribuye la solicitud de producto a los asistentes antes de la fecha de reunión.



Antes de una reunión FAST confecciona una lista de objetos, servicios, restricciones y criterios de rendimiento

Mientras se revisa la solicitud días antes de la reunión, se pide a todos los asistentes que hagan una lista de *objetos* que formen parte del entorno del sistema, otros objetos que debe producir el sistema y objetos que usa el sistema para desarrollar sus funciones. Además, se pide a todos los asistentes que hagan otra lista de *servicios* (procesos o funciones) que manipulan o interactúan con los objetos. Finalmente, se desarrollan también listas de *restricciones* (por ejemplo, costes, tamaño, peso) y criterios de rendimiento (por ejemplo, veloci-

[2] Dos de los enfoques más populares de TFEA son Desarrollo Conjunto de Aplicaciones (JAD), desarrollado por IBM, y The METHOD, desarrollado por Performance Resources, Inc., Falls Church, VA.

dad, precisión). Se informa a los asistentes que no se espera que las listas sean exhaustivas, pero que **sí** que reflejen su punto de vista del sistema.

Por ejemplo[3], imagínese que a un equipo de trabajo **TFEA** de una compañía de productos para el consumidor se le ha dado la siguiente descripción del producto:

> Nuestras investigaciones indican que el mercado de sistemas de seguridad para el hogar está creciendo a un ritmo del 40 por 100 anual. **Nos** gustaría entrar en este mercado construyendo un sistema de seguridad para el hogar basado en un microprocesador que proteja y/o reconozca varias situaciones indeseables tales como irrupciones ilegales, fuego, inundaciones y otras. El producto, provisionalmente llamado *HogarSeguro*, utilizará los sensores adecuados para detectar cada situación, puede programarse por el propietario del hogar y llamará automáticamente a una agencia de vigilancia cuando se detecte alguna de estas situaciones.

En realidad, se proporcionatía considerablemente más información en esta fase. Pero incluso con información adicional, habría ambigüedades presentes, existirán omisiones probablemente y podrían ocurrir errores. Por ahora, la «descripción de producto» anterior bastará.

El equipo TFEA está compuesto por representantes de marketing, ingeniería del software y del hardware, y de la fabricación también participará un coordinador externo.



los objetos deben ser manipulados por servicios y deben **«contemplar» las** restricciones y rendimientos definidos por el equipo FAST.

Todos los componentes del equipo TFEA desarrollan las listas descritas anteriormente. Los objetos descritos para *HogarSeguro* podrían incluir detectores de humo, sensores de ventanas y puertas, detectores de movimientos, una alarma, un acontecimiento (se ha activado un sensor), un panel de control, una pantalla, números de teléfono, una llamada de teléfono, etc. La lista de servicios podría incluir la instalación de la alarma, vigilancia de los sensores, marcado de teléfono, programación del panel de control y lectura de la pantalla (fíjese que los servicios actúan sobre los objetos). De igual manera, todos los asistentes TFEA desarrollarán una lista de restricciones (por ejemplo, el sistema debe tener un coste de fabricación de menos de £80, debe tener una «interfaz amigable» con el usuario y debe conectar directamente con una línea telefónica estándar) y de criterios de rendimiento (por ejemplo, un acontecimiento detectado por un sensor debe reconocerse en un segundo; se debería implementar un esquema de prioridad de acontecimientos).

Cuando empieza la reunión TFEA, el primer tema de estudio es la necesidad y justificación del nuevo producto, pues todo el mundo debería estar de acuerdo en que el desarrollo (o adquisición) del producto está justificada. Una vez que se ha conseguido el acuerdo, cada participante presenta **sus** listas para su estudio. Las listas pueden exponerse en las paredes de la habitación usando largas hojas de papel, pinchadas o pegadas, o escritas en una pizarra. Idealmente debería poderse manejar separadamente cada entrada de las listas para poder combinarlas, borrarlas o añadir otras. En esta fase, está estrictamente prohibido el debate o las críticas.



Evita el impulso de cortar la idea de un cliente por ((demasiado costosa» o «poco práctica». lo ideo es negociar alga que sea aceptable paro todos. Es decir, debes tener uno mente abierta.

Una vez que se han presentado las listas individuales sobre un tema, el grupo crea una lista combinada. La lista combinada elimina las entradas redundantes **y** añade las ideas nuevas que se les ocurran durante la presentación, pero no se elimina nada más. Cuando se han creado las listas combinadas de todos los temas, sigue el estudio —moderado por el coordinador—. La lista combinada es ordenada, ampliada o redactada de nuevo para reflejar apropiadamente el producto o sistema que se va a desarrollar. El objetivo es desarrollar una *lista **de** consenso* por cada tema (objetos, servicios, restricciones y rendimiento). Después estas listas se ponen aparte para utilizarlas posteriormente.

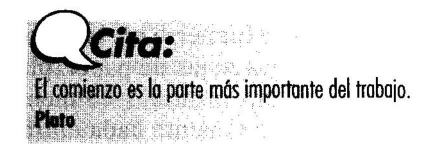
Una vez que se han completado las listas de consenso, el equipo se divide en subequipos; cada uno trabaja para desarrollar una mini-especificación de una o más entradas de cada lista[4]. La mini-especificación es una elaboración de la palabra o frase contenida en una lista. Por ejemplo, la mini-especificación del objeto **panel de control** de *HogarSeguro* podría ser: montado en la pared; tamaño aproximado 23 * 13 centímetros; contiene el teclado estándar de 12 teclas y teclas especiales; contiene una pantalla LCD de la forma mostrada en el bosquejo (no presentado aquí); toda la interacción del cliente se hace a través de las teclas usadas para activar o desactivar el sistema; software para proporcionar una directriz de empleo, recordatorios, etc., conectado a todos los sensores.

Cada subequipo presenta entonces **sus** mini-especificaciones a todos los asistentes TFEA para estudiarlas. Se realizan nuevos añadidos, eliminaciones y se sigue elaborando. En algunos casos, el desarrollo de algunas mini-especificaciones descubrirá nuevos objetos, servicios, restricciones o requisitos de rendimiento que se añadirán a las listas originales. Durante todos los estu-

[3] Este ejemplo (con extensiones y variaciones) se empleará para ilustrar métodos importantes de ingeniería del software en muchos de los capítulos siguientes. Como ejercicio, sería útil que realizara su propia reunión TFEA y desarrollara un conjunto de listas.

[4] Una alternativa puede ser la creación de casos **de** uso. Ver Sección 11.2.4 para más detalles.

dios, el equipo sacará a relucir aspectos que no podrán resolverse durante la reunión. Se hará una lista de estas ideas para tratarlas más adelante.



Una vez que se han completado las mini-especificaciones, cada asistente de la TFEA hace una lista de *criterios de validación* del producto/sistema y presenta su lista al equipo. Se crea así una lista de consenso de criterios de validación. Finalmente, uno o más participantes (o algún tercero) es asignado para escribir el borrador entero de especificación con todo lo expuesto en la reunión TFEA.

TFEA no es la panacea para los problemas que se encuentran en las primeras reuniones de requisitos, pero el enfoque de equipo proporciona las ventajas de muchos puntos de vista, estudio y refinamiento instantáneo **y** un paso adelante hacia el desarrollo de una especificación.

11.2.3. Despliegue de la función de calidad

El *despliegue de la función calidad (DFC)* es una técnica de gestión de calidad que traduce las necesidades del cliente en requisitos técnicos del software. Originalmente desarrollado en Japón y usado por primera vez en Kobe Shipyard of Mitsubishi Heavy Industries, Ltd. A primeros de los años 70, DFC «se concentra en maximizar la satisfacción del cliente» [ZUL92]. Para conseguirlo, DFC hace énfasis en entender lo que resulta valioso para el cliente y después desplegar estos valores a lo largo del proceso de ingeniería. DFC identifica tres tipos de requisitos [ZUL92]:



DFC define los requisitos orientados a **maximizar** la **satisfacción** del cliente.

Requisitos normales. Se declaran objetivos y metas para un producto o sistema durante las reuniones con el cliente. Si estos requisitos están presentes, el cliente quedará satisfecho. Ejemplos de requisitos normales podrían ser peticiones de tipos de presentación gráfica, funciones específicas del sistema y niveles definidos de rendimiento.

Requisitos esperados. Estos requisitos son implícitos al producto o sistema y pueden ser tan fundamentales que el cliente no los declara explícitamente. Su ausencia sería motivo de una insatisfacción significativa. Ejemplos de requisitos esperados **son:** facilidad de interacción hombre-máquina, buen funcionamiento y fiabilidad general, y facilidad de instalación del software.

Requisitos innovadores. Estas características van más allá de las expectativas del cliente y suelen ser muy satisfactorias. **Por** ejemplo, se pide un software procesador de textos con las características estándar. El producto entregado contiene ciertas capacidades de diseño de página que resultan muy válidas y que no eran esperadas.



Todos queremos ***implementar** muchos* requisitos ***apasionantes,*** pero debemos ser ***cuidadosos.*** Podemos concretar ***«requisitos inútiles»,*** o conducir *a* requisitos ***innovadores*** que conducen *a un* producto demasiado ***futurista.***

En realidad, el DFC se extiende a todo el proceso de ingeniería [AKA90]. Sin embargo, muchos conceptos DFC son aplicables al problema de la comunicación con el cliente a que se enfrenta un ingeniero del software durante las primeras fases del análisis de requisitos. Presentamos una visión general sólo de estos conceptos (adaptados al software de computadora) en los párrafos siguientes.

En las reuniones con el cliente el *despliegue de función* se emplea para determinar el valor de cada función requerida para el sistema. El *despliegue **de** información* identifica tanto los objetos de datos como los acontecimientos que el sistema debe producir y consumir. Estos están unidos a las funciones. Finalmente, el *despliegue de tareas* examina el comportamiento del sistema o producto dentro del contexto de su entorno. El *análisis de valor* es llevado a cabo para determinar la prioridad relativa de requisitos determinada durante cada uno de los tres despliegues mencionados anteriormente.



El Instituto DFC es una excelente fuente de información:
www.qfdi.org

El DFC utiliza observaciones y entrevistas con el cliente, emplea encuestas y examina datos históricos (por ejemplo, informes de problemas) como datos de base para la actividad de recogida de requisitos. Estos datos se traducen después en una tabla de requisitos —denominada tabla de opinión del cliente— que se revisa con el cliente. Entonces se usa una variedad de diagramas, matrices y métodos de evaluación para extraer los requisitos esperados e intentar obtener requisitos innovadores [BOS91].

11.2.4. Casos de uso

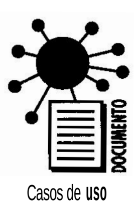
Una vez recopilados los requisitos, bien por reuniones informales, TFEA o DFC, el ingeniero del software (analista) puede crear un conjunto de escenarios que identifiquen una línea de utilización para el sistema que va a ser construido. Los escenarios, algunas veces llamados *casos de uso* [JAC92], facilitan una descripción de cómo el sistema se usará.



Un caso de uso es un escenario que describe cómo el **software** va a ser usado en uno determinada situación.

Para crear un caso de uso, el analista debe primero identificar los diferentes tipos de personas (o dispositivos) que utiliza el sistema o producto. Estos *actores* actualmente representan papeles que la gente (o dispositivos) juegan como impulsores del sistema. Definido **más** formalmente, un actor es algo que comunica con el sistema o producto y que es externo al sistema en sí mismo.

Es importante indicar que un actor y un usuario no son la misma cosa. Un usuario normal puede jugar un número de papeles diferentes cuando utiliza un sistema, por lo tanto un actor representa una clase de entidades externas (a veces, pero no siempre personas) que lleva a cabo un papel. Como ejemplo, considerar un operador de una máquina (un usuario) que interactúa con el ordenador central para un elemento de fabricación que contiene un número de robots y máquinas bajo control numérico. Después de una revisión cuidadosa de los requisitos, el software del computador central requiere cuatro modelos diferentes (papeles) de interacción: modo programación, modo prueba, modo monitorización y modo investigación. Además, se pueden definir cuatro actores: programador, probador, supervisor e investigador. En algunos casos, el operador de la máquina puede realizar todos los papeles. En otras ocasiones, diferentes personas pueden jugar el papel de cada actor.



Casos de **uso**

Porque la obtención de requisitos es una actividad de evolución, no todos los actores se identifican durante la primera iteración. Es posible identificar actores iniciales [JAC93] durante la primera iteración y actores secundarios cuando más se aprende del sistema. Los actores iniciales interactúan para conseguir funciones del sistema y derivar el beneficio deseado del sistema. Ellos trabaian directa **y** frecuentemente con el software. Los actores secundarios existen para dar soporte al sistema que los actores iniciales han dado forma con su trabajo.

Una vez que se han identificado los actores, se pueden desarrollar los casos de uso. El caso de uso describe la manera en que los actores interactúan con el sistema. Jacobson [JAC93] sugiere un número de preguntas que deberán responderse por el caso de uso:

- ¿Cuáles son las principales tareas o funciones que serán realizadas por el actor?
- ¿Cuál es el sistema de información que el actor adquiere, produce o cambia?
- ¿Qué actor informará al sistema de los cambios en el entorno externo?
- ¿Qué información necesita el actor sobre el sistema?



tos cosos de usos están definidos desde el punto de vista de un actor. Un actor es un **papel** que las personas (usuarios) o dispositivos juegan cuando interaccionon con el software.

En general, un caso de uso es, simplemente, un texto escrito que describe el papel de un actor que interactúa con el acontecer del sistema.

Volviendo a los requisitos básicos de *HogarSeguro* (Sección 11.2.2), podemos identificar tres actores: el propietario (el usuario), sensores (dispositivos vinculados al sistema), y el subsistema de monitorización y respuesta (la estación central que monitoriza *HogarSeguro*). Para los propósitos de este ejemplo, consideremos únicamente al actor **propietario.** El propietario interactúa con el producto en un número de diferentes caminos:

- introduce una contraseña para permitir cualquier interacción
- pregunta acerca del estado de una zona de seguridad
- pregunta acerca del estado de un sensor
- presiona el botón de alarma en caso de emergencia
- activa/desactiva el sistema de seguridad



Una **discusión** detallada de los casos de usos, incluyendo ejemplos, referencias y plontillos, se presenta en **members.aol.com/acockburn/papers/OnUseCases.htm**

Un caso de uso para el *sistema de activación* persigue:

1. El propietario observa un prototipo del panel de control de *HogarSeguro* (Fig. 11.2) para determinar si el sistema está preparado para la entrada. Si el sistema no está preparado, el propietario debe físicamente cerrar ventanas/puertas para que se encienda el indicador de preparado. (El indicador no preparado implica que el sensor está abierto, por ejemplo, que la puerta o ventana está abierta.)



FIGURA 11.2. Panel de control de *Hogar Seguro.*

2. El propietario utiliza el teclado para introducir una contraseña de cuatro dígitos. La contraseña es com-

parada con la contraseña válida almacenada en el sistema. Si la contraseña es errónea, el panel de control emitirá un sonido y se restaurará para incorporar una nueva entrada.

3. El propietario selecciona y pulsa *stay* o *away* (ver Fig. 11.2) para activar el sistema. *Stay* activa solamente sensores perimetrales (cuando detectan movimiento interno se desactivan). *Away* activa todos los sensores.
4. Cuando ocurre una activación, una luz de alarma puede ser observada por el propietario.

Cada caso de uso facilita un escenario sin ambigüedad en la interacción entre el actor y el software. Esto puede usarse para especificar requisitos de tiempo u otras restricciones del escenario. Por ejemplo, en el caso de uso referido anteriormente, los requisitos indican que ocurrirá 30 segundos después de pulsar la tecla *stay* o *away*. Esta información puede añadirse al caso de uso.

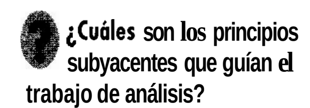
Los casos de uso describen escenarios que serán percibidos de distinta forma por distintos actores. Wyder [WYD96] indica que la calidad de la función desplegada puede ser usada para desarrollar un amplio valor de prioridades para cada caso de uso. Para acabar, los casos de uso son evaluados desde el punto de vista de todos los actores definidos para el sistema. Se asigna un valor de prioridad a cada caso de uso (por ejemplo, un valor de 1 a 10)para cada acto?. Se calcula una prioridad, indicando la importancia percibida de cada caso de uso.

Cuando un modelo de proceso iterativo es usado por la ingeniería del software orientado a objetos, las prioridades pueden influir en qué funcionalidad del sistema será entregada primero.

11.3 PRINCIPIOS DEL ANÁLISIS

Durante las dos pasadas décadas, se han desarrollado un gran número de métodos de modelado. Los investigadores han identificado los problemas del análisis y **sus** causas y han desarrollado varias notaciones de modelado y sus correspondientes conjuntos de heurísticas para solucionarlos. Cada método de análisis tiene **su** punto de vista. Sin embargo, todos los métodos de análisis se relacionan por un conjunto de principios operativos:

1. Debe representarse y entenderse el dominio de información de un problema.
2. Deben definirse las funciones que debe realizar el software.
3. Debe representarse el comportamiento del software (como consecuencia de acontecimientos externos).
4. Deben dividirse los modelos que representan información, función y comportamiento de manera que se descubran los detalles por capas *(o* jerárquicamente).
5. El proceso de análisis debería ir desde la información esencial hasta el detalle de la implementación.


¿Cuáles son los principios subyacentes que guían el trabajo de análisis?

Aplicando estos principios, el analista se aproxima al problema sistemáticamente. Se examina el dominio de información de manera que pueda entenderse completamente la función. Se emplean modelos para poder comunicar de forma compacta las características de la función y **su** comportamiento. Se aplica la partición para reducir la complejidad. Son necesarias las visiones esenciales y de implementación del software para acomodar las restricciones lógicas impuestas por los requisitos del procesamiento y las restricciones físicas impuestas por otros elementos del sistema.

Además de los principios operativos de análisis mencionados anteriormente, Davis [DAV95a] sugiere un conjunto[6] de principios directrices para la «ingeniería de requisitos»:

- *Entender el problema antes de empezar a crear el modelo de análisis.* Hay tendencia a precipitarse en busca de una solución, incluso antes de entender el problema. ¡Esto lleva a menudo a un elegante software para el problema equivocado!
- *Desarrollar prototipos que permitan al usuario entender cómo será la interacción hombre-máquina.* Como el concepto de calidad del software se basa a menudo en la opinión sobre la «amigabilidad» de la interfaz, el desarrollo de prototipos (y la iteración que se produce) es altamente recomendable.
- *Registrar el origen y la razón de cada requisito.* Este es el primer paso para establecer un seguimiento hasta el cliente.
- *Usar múltiples planteamientos de requisitos.* La construcción de modelos de datos, funcionales y de comportamiento, le proporcionan al ingeniero del software tres puntos de vista diferentes. Esto reduce la probabilidad de que se olvide algo y aumenta la probabilidad de reconocer la falta de consistencia.
- *Dar prioridad a los requisitos.* Las fechas ajustadas de entrega pueden impedir la implementación de

[5] Lo ideal es que la evaluación sea realizada por funciones individuales de la organización o negocio representadas por un actor.

[6] Aquí se menciona sólo un pequeño subconjunto de los principios de ingeniería de requisitos de Davis. Para obtener más información, véase [DAV95a].

todos los requisitos del software. Si se aplica un modelo de proceso incremental (Capítulo 2), se deben identificar los requisitos que se van a entregar en la primera entrega.

- *Trabajar para eliminar la ambigüedad.* Como la mayoría de los requisitos están descritos en un lenguaje natural, abunda la oportunidad de ambigüedades. El empleo de revisiones técnicas formales es una manera de descubrir y eliminar la ambigüedad.



Un computador hará lo que le digas, pero ello puede ser muy diferente de lo que tengas en mente.
Joseph Weizenbaum

Un ingeniero del software que se aprenda estos principios de memoria es muy probable que desarrolle una especificación del software que proporcione un excelente fundamento para el diseño.

11.3.1. El dominio de la información

Todas las aplicaciones software pueden denominarse colectivamente *procesamiento de datos.* Es interesante que este término contenga una clave para nuestro entendimiento de los requisitos del software. El software se construye para procesar datos, para transformar datos de una forma a otra, es decir, para aceptar una entrada de información, manipularla de alguna manera y producir una salida de información. Esta declaración fundamental de objetivos es cierta, tanto si construimos software por lotes para un sistema de nóminas, como si construimos software dedicado de tiempo real para controlar el flujo de combustible de un motor de automóvil.



El dominio de información de un problema agrupan elementos de datos u objetos que contienen números, texto, imágenes audio, video o cualquier combinación de ellos.

Es importante recalcar, sin embargo, que el software también procesa sucesos. Un suceso representa algún aspecto de control del sistema y no es más que un dato **binario** (es encendido o apagado, verdadero o falso, está allí o no). Por ejemplo, un sensor de presión detecta la presión que excede de un valor seguro y envía una señal de alarma al software de vigilancia. La señal de alarma *es* un suceso que controla el comportamiento del sistema. Por tanto, los datos (números, caracteres, imágenes, sonidos, etc.) y el control (acontecimientos) residen dentro del dominio de la información de un problema.

El primer principio operativo de análisis requiere el examen del dominio de la información y la creación de un *modelo de datos.* Este dominio contiene tres visiones diferentes de los datos y del control a medida que se procesa cada uno en un programa de computadora: (1) contenido de la información y relaciones, (2) flujo de la información y (3) estructura de la información. Para entender completamente el dominio de la información, debería estudiarse cada una de estas visiones.



Para comenzar a comprender el dominio de información, la primera pregunta que debe ser realizada es: «¿Qué información de salida generará el sistema?»

El contenido de la información representa los objetos individuales de datos y de control que componen alguna colección mayor de información a la que transforma el software. Por ejemplo, el objeto datos **cheque** es una composición de varios componentes de información importantes: el nombre del beneficiario, la cantidad neta a pagar, el importe bruto, deducciones, etc. Por tanto, el contenido de **cheque** es definido por los atributos necesarios para crearlo. De forma similar, el contenido de un objeto de control denominado **estado del sistema** podría definirse mediante una cadena de bits. Cada bit representa un elemento diferente de información que indica si un dispositivo en particular está en línea o fuera de línea.

Los objetos de datos y de control pueden relacionarse con otros objetos de datos o de control. Por ejemplo, el objeto de datos cheque tiene una o más relaciones con los objetos empleado, banco y otros. Durante el análisis del dominio de la información, se deberían definir estas relaciones.

El flujo de la información representa cómo cambian los datos y el control a medida que se mueven dentro de un sistema. Como se muestra en la Figura 11.3, los objetos de entrada se transforman para intercambiar información (datos y/o control), hasta que se transforman en información de salida. A lo largo de este camino de transformación (o caminos), se puede introducir información adicional de un almacén de datos (por ejemplo, un archivo en disco o memoria intermedia). Las transformaciones que se aplican a los datos son funciones o subfunciones que debe realizar un programa. Los datos y control que se mueven entre dos transformaciones (funciones) definen la interfaz de cada función.

La *estructura de la información* representa la organización interna de los elementos de datos o de control. ¿Hay que organizar los elementos de datos o de control como una tabla de dimensión *n* o como una estructura jerárquica en árbol? Dentro del contexto de la estructura, ¿qué información está relacionada con otra información? ¿Está contenida toda la información en una sola estructura o se van a utilizar varias? ¿Cómo se relaciona la información de una estructura con la de otra? Estas y otras preguntas se responden mediante una valoración de la estructura de la información.

Debería resaltase que la estructura de datos, un concepto afín que se estudiará más adelante en este libro, se refiere al diseño e implementación de la estructura de la información.

11.3.2. Modelado

Los modelos se crean para entender mejor la entidad que se va a construir. Cuando la entidad es algo físico (un edificio, un avión, una máquina), podemos construir un modelo idéntico en forma, pero más pequeño. Sin embargo, cuando la entidad que se va a construir es software, nuestro modelo debe tomar una forma diferente. Debe ser capaz de modelar la información que transforma el software, las funciones (y subfunciones) que permiten que ocurran las transformaciones y el comportamiento del sistema cuando ocurren estas transformaciones.

El segundo y tercer principios operativos del análisis requieren la construcción de modelos de función y comportamiento.

¿ Qué tipos de modelos debemos crear durante el análisis de requisitos?

Modelos funcionales. El software transforma la información y, para hacerlo, debe realizar al menos tres funciones genéricas: entrada, procevamiento y salida. Cuando se crean modelos funcionales de una aplicación, el ingeniero del software se concentra en funciones específicas del problema. El modelo funcional empieza con un sencillo modelo a nivel de contexto (por ejemplo, el nombre del software que se va a construir). Después de una serie de iteraciones, se consiguen más y más detalles funcionales, hasta que se consigue representar una minuciosa definición de toda la funcionalidad del sistema.

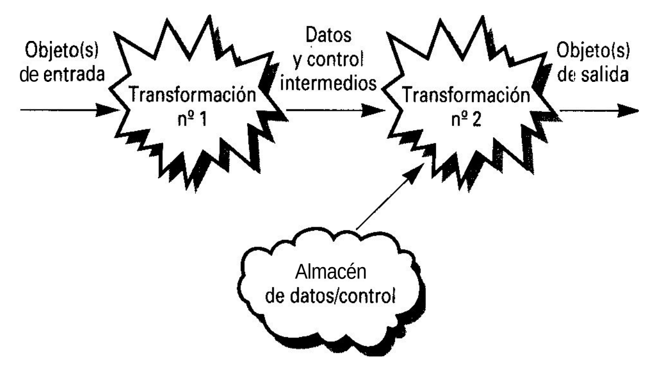
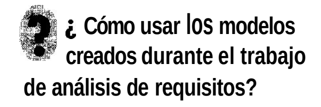


FIGURA 11.3. Flujo y transformación de la información.

Modelos de comportamiento. La mayoría del software responde a los acontecimientos del mundo exterior. Esta característica estímulo-respuesta forma la base del modelo del comportamiento. Un programa de computadora siempre está en un estado, un modo de comportamiento observable exteriormente (por ejemplo, esperando, calculando, imprimiendo, haciendo cola) que cambia sólo cuando ocurre algún suceso. Por ejemplo, el software permanecerá en el estado de espera hasta que (1) un reloj interno le indique que ha pasado cierto intervalo de tiempo, (2) un suceso externo (por ejemplo, un movimiento del ratón) provoca una interrupción, o (3) un sistema externo señala al software que actúe de alguna manera. Un modelo de comportamiento crea una representación de los estados del software y los sucesos que causan que cambie de estado.

Los modelos creados durante el análisis de requisitos desempeñan unos papeles muy importantes:

- El modelo ayuda al analista a entender la información, la función y el comportamiento del sistema, haciendo por tanto más fácil y sistemática la tarea de análisis de requisitos.
- El modelo se convierte en el punto de mira para la revisión y por tanto la clave para determinar si se ha completado, su consistencia y la precisión de la especificación.
- El modelo se convierte en el fundamento para el diseño, proporcionando al diseñador una representación esencial del software que pueda trasladarse al contexto de la implementación.


¿ Cómo usar los modelos creados durante el trabajo de análisis de requisitos?

Los métodos de análisis estudiados en los Capítulos 12 y 21 son de hecho métodos de modelado. Aunque el método de modelado empleado es a menudo cuestión de preferencias personales (o de la organización), la actividad de modelado es fundamental para un buen trabajo de análisis.

11.3.3. Partición

A menudo los problemas son demasiado grandes o complejos para entenderlos globalmente. Por este motivo, tendemos a hacer una partición (dividir) de estos problemas en partes que puedan entenderse fácilmente y establecer las interacciones entre las partes de manera que se pueda conseguir la función global. El cuarto principio operativo del análisis sugiere que se pueden partir los dominios de la información, funcional y de comportamiento.



La descomposición es un proceso que resulta de la elaboración de datos, funciones o comportamientos. Puede ser realizada horizontal o verticalmente,

En esencia, la *partición* descompone un problema en sus partes constitutivas. Conceptualmente, establecemos una representación jerárquica de la información o de la función y después partimos el elemento de orden superior (1) exponiendo más detalles cada vez al movernos verticalmente en la jerarquía o (2) descomponiendo el problema si nos movemos horizontalmente en la jerarquía. Para ilustrar estos enfoques de partición, reconsideremos el sistema de seguridad *HogarSeguro* descrito en la Sección 11.2.2. La asignación de software para *HogarSeguro* (obtenido como consecuencia de las acti-

vidades de ingeniería del sistema y de las reuniones TFEA) puede establecerse en los párrafos siguientes:

> **El** software de *HogarSeguro* permite al propietario configurar el sistema de seguridad cuando se instala, vigila todos los sensores conectados al sistema de seguridad e interactúa con el propietario a través de un teclado y teclas funcionales del panel de control de *HogarSeguro* mostrado en la Figura **11.2.**
>
> Durante la instalación, el panel de control de *HogarSeguro* se emplea para «programar» y configurar el sistema. **A** cada sensor se le asigna **un** número y un tipo, se programa una contraseña para activar y desactivar el sistema y se introducen el (los) número(s) de teléfono que se marcará(n) cuando un sensor detecte un suceso que haga saltar **la** alarma.
>
> Cuando un sensor detecta un acontecimiento, el software invoca una alarma sonora asociada al sistema. Después de un retardo especificado por el propietario durante la configuración del sistema, el software marca un número de teléfono de una empresa de seguridad y proporciona información sobre **la** dirección, informando de la naturaleza del acontecimiento detectado. El número se volverá **a** marcar cada 20 segundos hasta que **se** obtenga la conexión telefónica.
>
> Toda la interacción con *Hogarseguro* es manejada por un subsistema de interacción con el usuario que lee la entrada de información proporcionada **a través** del teclado y las teclas funcionales, las pantallas que presentan mensajes y el estado del sistema en **la** pantalla LCD. **La** interacción con el teclado toma la siguiente forma...

Los requisitos del software de *HogarSeguro* pueden analizarse partiendo los dominios de información, funcional y de comportamiento del producto. Para ilustrarlo, partiremos el dominio funcional del problema. La Figura 11.4 ilustra una *descomposición horizontal* del software *HogarSeguro.* El problema se parte mediante la representación de las funciones constitutivas del software *HogarSeguro,* moviéndose horizontalmente en la jerarquía funcional. En el primer nivel de la jerarquía aparecen tres funciones principales.

Cita:
El comprender no puede ser sustituido por una actividad intensa.
H.H. Williams

Las subfunciones asociadas con una función principal de *HogarSeguro* pueden examinarse mediante la exposición vertical detallada en la jerarquía, tal y como se ilustra en la Figura 11.5. Moviéndose hacia abajo a lo largo de **un** camino, por ejemplo la función sensores de vigilancia, la partición se hace vertical para mostrar mayores niveles de detalle funcional. El enfoque de partición que **hemos** aplicado a las funciones de *HogarSeguro* puede aplicarse también al dominio de la información y al de comportamiento. De hecho, la partición del flujo de la información y del comportamiento del sistema (tratados en el Capitulo 12) proporcionará una visión más profunda de los requisitos del sistema. **A** medida que se parte el sistema, se obtienen las interfaces entre las funciones. Los datos y elementos de control que se mueven a través de una interfaz deberían restringirse a las entradas necesarias para realizar la función en cuestión y a las salidas requeridas por otras funciones o elementos del sistema.

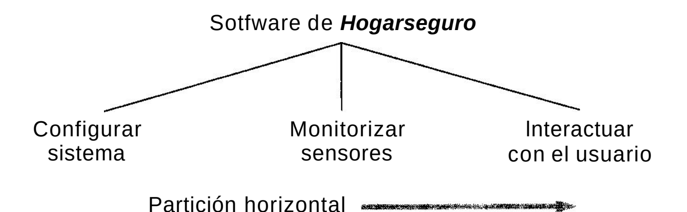


FIGURA 11.4. Partición horizontal de una función de *Hogar Seguro.*

11.3.4. Visiones esenciales y de implementación[7]

Una *visión esencial* de los requisitos del software presenta las funciones a conseguir y la información a procesar sin tener en cuenta los detalles de la implementación. Por ejemplo, la visión esencial de la función de *HogarSeguro leer el estado del sensor* no tiene nada que ver con la forma física de los datos o el tipo de sensor que se emplea. De hecho, se podría argumentar que leer estado sería un nombre más apropiado para esta función, ya que ignora totalmente los detalles del mecanismo de entrada. Igualmente, un modelo de datos esencial del elemento datos **número de teléfono** (implicado en la función *marcar número de teléfono)* puede representarse en esta fase independientemente de la estructura de los datos (si es que hay alguna) usada para implementar el elemento datos. Enfocando nuestra atención en la esencia del problema en las primeras fases del análisis de requisitos, dejamos abiertas nuestras opciones para especificar los detalles de implementación durante fases posteriores de especificación de requisitos y diseño del software.



*Evita la tentación de **trasladarte** directamente al punta de vista de implementación, entendiendo que lo esencia del problema es obvia. El especificar demasiado rápido la implementación en detalle reduce **tus alternativas** e incremento el riesgo.*

La *visión de implementación* de los requisitos del software introduce la manifestación en el mundo real de las funciones de procesamiento y las estructuras de la información. En algunos casos, se desarrolla una representación física en la primera fase del diseño del software. Sin embargo, la mayoría de los sistemas basados en computadora se especifican de manera que

[7] Mucha gente utiliza términos visión «lógica» y «física» para referirse al mismo concepto.

se acomode a ciertos detalles de implementación. Un dispositivo de entrada de información a *HogarSeguro* podría ser un sensor de perímetro (no un perro guardián, un guardia de seguridad o una trampa). El sensor detecta una entrada no autorizada al detectar una ruptura en un circuito electrónico. Las características generales del sensor deberían tenerse en cuenta como parte de la especificación de los requisitos del software. El analista debe reconocer las restricciones impuestas por elementos predefinidos del sistema (el sensor) y considerar la visión de implementación de la función y de la información cuando tal visión es apropiada.

Ya hemos comentado anteriormente que el análisis de los requisitos del software debería concentrarse en qué es lo que debe hacer el software, en vez de cómo se implementará el procesamiento. Sin embargo, la visión de implementación no debería interpretarse necesariamente como una representación del cómo. **Más** bien, un modelo de implementación representa el modo actual de operación, es decir, la asignación existente o propuesta de todos los elementos del sistema. El modelo esencial (de función o datos) es genérico en el sentido de que la realización de la función no se indica explícitamente.

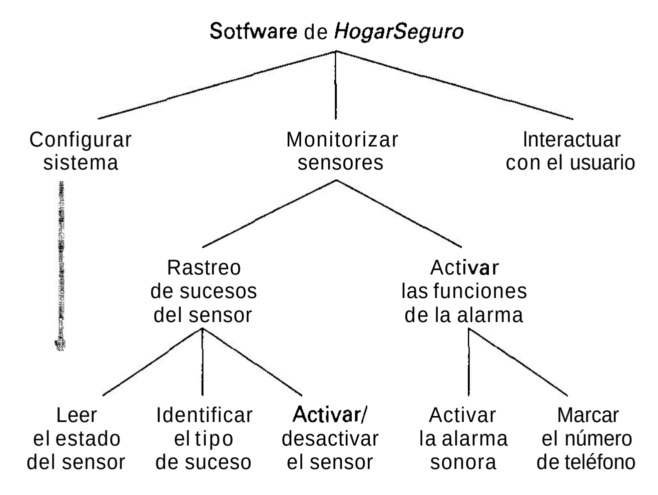
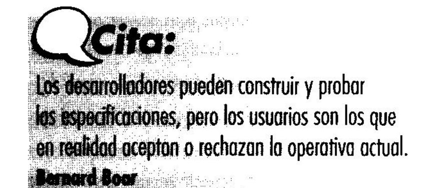


FIGURA 11.5. Partición vertical de la función de *HogarSeguro.*

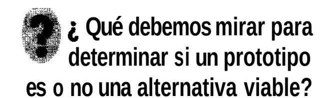
11.4 CREACIÓN DE PROTOTIPOS DEL SOFTWARE

El análisis hay que hacerlo independientemente del paradigma de ingeniería del software que se aplique. Sin embargo, la forma que toma este análisis varía. En algunos casos es posible aplicar los principios operativos del análisis y obtener un modelo de software del que se pueda desarrollar un diseño. En otras situaciones, se realizan recopilaciones de requisitos (por TFEA, DFC u otras técnicas de «tormenta de ideas» [JOR89]), se aplican los principios del análisis y se construye un modelo del software a fabricar denominado prototipo para que lo valore el cliente y el desarrollador. Finalmente, hay circunstancias que requieren la construcción de un prototipo al comienzo del análisis, ya que el modelo es el único medio a través del cual se pueden obtener eficazmente los requisitos. El modelo evoluciona después hacia la producción del software.



11.4.1. Selección del enfoque de creación de prototipos

El paradigma de creación de prototipos puede ser cerrado o abierto. El enfoque cerrado se denomina a menudo *prototipo desechable*. Este prototipo sirve únicamente como una basta demostración de los requisitos. Después se desecha y se hace una ingeniería del software con un paradigma diferente. Un enfoque abierto, denominado *prototipo evolutivo*, emplea el prototipo como primera parte de una actividad de análisis a la que seguirá el diseño y la construcción. El prototipo del software es la primera evolución del sistema terminado.



Antes de poder elegir un enfoque abierto o cerrado, es necesario determinar si se puede crear un prototipo del sistema a construir. Se pueden definir varios factores candidatos a la creación de prototipos [BOA84]: área de aplicación, complejidad, características del cliente y del proyecto[8].

En general, cualquier aplicación que cree pantallas visuales dinámicas, interactúe intensamente con el ser humano, o demande algoritmos o procesamiento de combinaciones que deban crearse de manera progresiva, es un buen candidato para la creación de un prototipo. Sin embargo, estas áreas de aplicación deben ponderarse con la complejidad de la aplicación. Si una aplicación candidata (una con las características reseñadas anteriormente) va a requerir el desarrollo de decenas de miles de líneas de código

[8] Se puede encontrar otro útil estudio sobre los factores candidatos de «cuando hacer un prototipo» en [DAV95b].

antes de poder realizar cualquier función demostrable, es muy probable que sea demasiado compleja para crear un prototipo[9]. Si se puede hacer una partición a su complejidad, puede ser posible crear prototipos de porciones del software.

Como el cliente debe interactuar con el prototipo en fases posteriores, es esencial que (1) se destinen recursos del cliente a la evaluación y refinamiento del prototipo, y (2) el cliente sea capaz de tomar decisiones inmediatas sobre los requisitos. Finalmente, la naturaleza del proyecto de desarrollo tendrá una gran influencia en la capacidad de crear un prototipo. ¿Desea y es capaz la gestión del proyecto de trabajar con el método del prototipo? ¿,Tenemos disponibles herramientas para la creación de prototipos? ¿Tienen experiencia los desarrolladores con los métodos de creación de prototipos? Andriole [AND92] sugiere un conjunto de seis preguntas (Fig. 11.6) e indica unos conjuntos básicos de respuestas con su correspondiente tipo de prototipo sugerido.

11.4.2. Métodos y herramientas para el desarrollo de prototipos

Para que la creación del prototipo de software sea efectivo, debe desarrollarse rápidamente para que el cliente pueda valorar los resultados y recomendar los cambios oportunos. Para poder crear prototipos rápidos, hay disponibles tres clases genéricas de métodos y herramientas (por ejemplo, [AND92], [TAN89]):

Técnicas de Cuarta Generación. *Las técnicas de cuarta generación (T4G)* comprenden una amplia gama de lenguajes de consulta e informes de bases de datos, generadores de programas y aplicaciones y de otros lenguajes no procedimentales de muy alto nivel. Como las técnicas T4G permiten al ingeniero del software generar código ejecutable rápidamente, son ideales para la creación rápida de prototipos.

Componentes de software reutilizables. Otro enfoque para crear prototipos rápidos es ensamblar, más que construir, el prototipo mediante un conjunto de componentes software existentes. La combinación de prototipos con la reutilización de componentes de programa sólo funcionará si se desarrolla un sistema bibliotecario de manera que **los** componentes existentes estén catalogados **y** puedan recogerse. Debería resaltarse que se puede usar **un** prodiicto software existente como prototipo de un «nuevo **y** mejorado» producto competitivo. En cierta manera, ésta es una forma de reutilización en la creación de prototipos software.

Especificaciones formales **y** entornos para prototipos. Durante las pasadas dos décadas, se han desarrollado varios lenguajes formales de especificación y herramientas como sustitutos de las técnicas de especificación con lenguaje natural. Hoy en día, los desarrolladores de estos lenguajes formales están desarrollando entornos interactivos que (1) permitan al analista crear interactivamente una especificación basada en lenguaje de un sistema o software, (2) invoquen herramientas automáticas que traducen la especificación basada en el lenguaje en código ejecutable, y **(3)** permitan al cliente usar el código ejecutable del prototipo para refinar los requisitos formales.

Pregunta	Prototipo desechable	Prototipo evolutivo	Trabajo preliminar adicional requerido
¿Se entiende el dominio de la aplicación?	Sí	Sí	No
¿Se puede modelar el problema?	Sí	Sí	No
¿Está el cliente suficientemente seguro de los requisitos básicos del sistema?	Sí/No	Sí/No	No
¿Están establecidos los requisitos y son estables?	No	Sí	Sí
¿Hay requisitos ambiguos?	Sí	No	Sí
¿Hay contradicciones en los requisitos?	Sí	No	Sí

FIGURA 11.6. Selección del enfoque apropiado de creación de prototipo.

11.5 ESPECIFICACIÓN

No hay duda de que el modo de especificación tiene mucho que ver con la calidad de la solución. Los ingenieros del software que se han visto forzados a trabajar con especificaciones incompletas, inconsistentes o engañosas han experimentado la frustración y confusión que invariablemente provocan. La calidad, la fecha de entrega y el alcance del software son las que sufren las consecuencias.



En muchos casos, no es razonable plantearse que la especificación deberá contrastarse con todo. Debemos capturar lo esencia de lo que el cliente solicito.

[9] En algunos casos, se pueden construir rapidamente prototipos extremadamente complejos usando técnicas de cuarta generacion o componentes software reutilizables

11.5.1. Principios de la especificación

La especificación, independientemente del modo como la realicemos, puede verse como un proceso de representación. Los requisitos se representan de manera que como fin último lleven al éxito de la implementación del software. A continuación, se proponen algunos principios de especificación adaptados del trabajo de Balzer y Goldman [BAL86]:

1. Separar la funcionalidad de la implementación.
2. Desarrollar un modelo del comportamiento deseado de un sistema que comprenda datos y las respuestas funcionales de un sistema a varios estímulos del entorno.
3. Establecer el contexto en que opera el software especificando la manera en que otros componentes del sistema interactúan con él.
4. Definir el entorno en que va a operar el sistema e indicar como «una colección de agentes altamente entrelazados reaccionan a estímulos del entorno (cambios de objetos) producidos por esos agentes» [BAL86].
5. Crear un modelo intuitivo en vez de un diseño o modelo de implementación.
6. Reconocer que «la especificación debe ser tolerante a un posible crecimiento si no es completa». Una especificación es siempre un modelo —una abstracción— de alguna situación real (o prevista) que normalmente suele ser compleja. De ahí que será incompleta y existirá a muchos niveles de detalle.
7. Establecer el contenido y la estructura de una especificación de manera que acepte cambios.

Esta lista de principios básicos proporciona la base para representar los requisitos del software. Sin embargo, los principios deben traducirse en realidad. En la siguiente sección examinamos un conjunto de directrices para la creación de una especificación de requisitos.

11.5.2. Representación

Ya hemos visto que los requisitos del software pueden especificarse de varias maneras. Sin embargo, si los requisitos se muestran en papel o en un medio electrónico de representación (¡y debería ser casi siempre así!), merece la pena seguir este sencillo grupo de directrices:

¿Cuáles son las directrices básicas fundamentales para representar los requisitos?

El formato de la representación y el contenido deberían estar relacionados con el problema. **Se puede desarrollar un esbozo general del contenido de la especificación de los requisitos del software. Sin embargo, las formas de representación contenidas en la especificación es probable que varíen con el área de aplicación. Por ejemplo, la especificación de un sistema automático de fabricación utilizaría diferente simbología, diagramas y lenguaje que la especificación de un compilador de un lenguaje de programación.**

La información contenida dentro de la especificación debería estar escalonada. **Las representaciones deberían revelar capas de información de manera que el lector se pueda mover en el nivel de detalle requerido. La numeración de párrafos y diagramas debería indicar el nivel de detalle que se muestra. A veces, merece la pena presentar la misma información con diferentes niveles de abstracción para ayudar a su comprensión.**

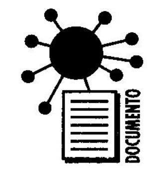
Los diagramas y otras formas de notación deberían restringirse en número y ser consistentes en su empleo. **Las notaciones confusas o inconsistentes, tanto gráficas como simbólicas, degradan la comprensión y fomentan errores.**

Las representaciones deben permitir revisiones. **Seguramente el contenido de una especificación cambiará. Idealmente, debería haber herramientas CASE disponibles para actualizar todas las representaciones afectadas por cada cambio.**

Los investigadores han llevado a cabo numerosos estudios (por ejemplo, [HOL95], [CUR85]) sobre los factores humanos asociados con la especificación. Parece haber pocas dudas de que la simbología y la distribución afectan a la comprensión. Sin embargo, los ingenieros del software parecen tener preferencias individuales por especificaciones simbólicas y diagramas específicos. La familiaridad es a menudo la raíz de las preferencias de una persona, pero otros factores más tangibles tales como la disposición espacial, formas fácilmente reconocibles y el grado de formalidad dictan normalmente la elección individual.

11.5.3. La especificación de los requisitos del software

La *especificación de los requisitos del software* se produce en la culminación de la tarea de análisis. La función y rendimiento asignados al software como parte de la ingeniería de sistemas se retinan estableciendo una completa descripción de la información, una descripción detallada de la función y del comportamiento, una indicación de los requisitos del rendimiento y restricciones del diseño, criterios de validación apropiados y otros datos pertinentes a los requisitos.



Especificación de Requisitos Software

La Introducción a la especificación de los requisitos del software establece las metas y objetivos del software, describiéndolo en el contexto del sistema basado en computadora. De hecho, la Introducción puede no ser más que el alcance del software en el documento de planificación.

La Descripción de la información proporciona una detallada descripción del problema que el software va a resolver. Se documentan el contenido de la información y sus relaciones, flujo y estructura. Se describen las interfaces hardware, software y humanas para los elementos externos del sistema y para las funciones internas del software.

En la *Descripción funcional* se describen todas las funciones requeridas para solucionar el problema. Se proporciona una descripción del proceso de cada función; se establecen y justifican las restricciones del diseño; se establecen las características del rendimiento; y se incluyen uno o más diagramas para representar gráficamente la estructura global del software y las interacciones entre las funciones software y otros elementos del sistema. La sección de las especificaciones *Descripción del comportamiento* examina la operativa del software como consecuencia de acontecimientos externos y características de control generadas internamente.



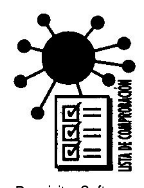
Cuando desorrolles criterios de ***validación,*** *responde a lo siguiente cuestión: «Cómo reconocer si un sistema es correcto si no lo muevo de mi mesa?»*

Probablemente la más importante, e irónicamente, la sección más a menudo ignorada de una especificación de requisitos del software son los *Criterios de validación.* ¿Cómo reconocemos el éxito de una implementación? ¿Qué clases de pruebas se deben llevar a cabo para validar la función, el rendimiento y las restricciones? Ignoramos esta sección porque para completarla se necesita una profunda comprensión de los requisitos del software, algo que a menudo no poseemos en esta fase. Sin embargo, la especificación de los criterios de validación actúa como una revisión implícita de todos los demás requisitos. **Es** esencial invertir tiempo y prestar atención a esta sección. Finalmente, la especificación incluye una *Bibliografía* y un *Apéndice.*

En muchos casos la *Especificación de requisitos del software* puede estar acompañada de un prototipo ejecutable (que en algunos casos puede sustituir a la especificación), un prototipo en papel o un *Manual de usuario preliminar. El Manual de usuario preliminar* presenta al software como una caja negra. Es decir, se pone gran énfasis en las entradas del usuario **y** las salidas (resultados) obtenidas. **El** manual puede servir como una valiosa herramienta para descubrir problemas en la interfaz hombre-máquina.

11.6 REVISIÓN DE LA ESPECIFICACIÓN

La revisión de la *Especificación de requisitos del software* (y/o prototipo) es llevada a cabo tanto por el desarrollador del software como por el cliente. Como la especificación forma el fundamento para el diseño y las subsiguientes actividades de la ingeniería del software, se debería poner extremo cuidado al realizar la revisión.



Requisitos Software
Revisión de la Especificación

Inicialmente la revisión se lleva a cabo a nivel macroscópico. A este nivel, los revisores intentan asegurarse de que la especificación sea completa, consistente y correcta cuando la información general, funcional y de los dominios del comportamiento son considerados. Asimismo, una exploración completa de cada uno de estos dominios, la revisión profundiza en el detalle, examinando no solo las descripciones superficiales, sino la vía en la que los requisitos son expresados. Por ejemplo, cuando una especificación contiene un «término vago» (por ejemplo, algo, algunas veces, a veces, normalmente, corrientemente, mucho, o principalmente), el revisor señalará la sentencia para su clarificación.

Una vez que se ha completado la revisión, se firma la especificación de requisitos del software por el cliente y el desarrollador. La especificación se convierte en un «contrato» para el desarrollo del software. Las peticiones de cambios en los requisitos una vez que se ha finalizado la especificación no se eliminarán, pero el cliente debe saber que cada cambio a posteriori significa una ampliación del ámbito del software, y por tanto pueden aumentar los costes y prolongar los plazos de la planificación temporal del proyecto.

Incluso con los mejores procedimientos de revisión, siempre persisten algunos problemas comunes de especificación. La especificación es difícil de «probar» de manera significativa, por lo que pueden pasar inadvertidas ciertas inconsistencias y omisiones. Durante la revisión, se pueden recomendar cambios a la especificación. Puede ser extremadamente difícil valorar el impacto global de un cambio; es decir, ¿cómo afecta el cambio en una función a los requisitos de las demás? Los modernos entornos de ingeniería del software (Capítulo 31) incorporan herramientas CASE desarrolladas para ayudar a resolver estos problemas.

RESUMEN

El análisis de requisitos es la primera fase técnica del proceso de ingeniería del software. En este punto se refina la declaración general del ámbito del software en una especificación concreta que se convierte en el fundamento de todas las actividades siguientes de la ingeniería del software.

El análisis debe enfocarse en los dominios de la información, funcional y de comportamiento del problema. Para entender mejor lo que se requiere, se crean modelos, los problemas sufren una partición y se desarrollan representaciones que muestran la esencia de los requisitos y posteriormente los detalles de la implementación.

En muchos casos, no es posible especificar completamente un problema en una etapa tan temprana. La creación de prototipos ofrece un enfoque alternativo que produce un modelo ejecutable del software en el que se pueden refinar los requisitos. Se necesitan herramientas especiales para poder realizar adecuadamente la creación de prototipos.

Como resultado del análisis, se desarrolla la ***Especificación de requisitos del software***. La revisión es esencial para asegurarse que el cliente y el desarrollador tienen el mismo concepto del sistema. Desgraciadamente, incluso con los mejores métodos, la cuestión es que el problema sigue cambiando.

REFERENCIAS

[AKA90] Akao, Y. (de.), *Quality Function Deployment: Integrating Customer Requirements in Product Design* (traducido por G. Mazur), Productivity Press, Cambridge MA, 1990

[BAL86] Balzer, R., y N. Goodman, «Principles of Good Specification and their Implications for Specification Languages», in *Software Specification Techniques* (Gehani and McGetrick, eds.), Addison-Wesley, 1986, pp. 25-39.

[BOA84] Boar, B., *Application Prototyping*, Wiley-Interscience, 1984.

[BOS91] Bossert, J.L., *Quality Function Deployment: A Practitioner's Approach*, ASQC Press, 1991.

[CUR85] Curtis, B., *Human Factors in Software Development*, IEEE Computer Society Press, 1985.

[DAV93] Davis, A., *Software requirements: Ohjetcs, Functions and Stutes*, Prentice-Hall, 1993.

[DAV95a] Davis, A., 201 *Principles of Software Development*, McGraw-Hill, 1995.

[DAV95b] Davis, A., «Software Prototyping», in *Advances in Computers*, vol. 40, Academic Press, 1995.

[GAU89] Gause, D.C., y G. M. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House, 1989.

[HOL95] Holtzblatt, K., y **E.** Carmel (eds.), «Requirements Gathering: The Human Factor», un resultado especial de CACM, vol. 38, n.° 5, Mayo 1995.

[JAC92] Jacobson, I., *Ohject-Oriented Software Engineering*, Addison-Wesley, 1992.

[JOR89] Jordan, P.W. et al., «Software Storming: Combining Rapid Prototyping and Knowledge Engineering», *IEEE Computer*, vol. 22, n.° 5, Mayo 1989, pp. 39-50.

[REI94] Reifer, D.J., «Requirements Engineering», in *Encyclopedia of Software Engineering* (J.J Marciniak, editor), Wiley, 1994, pp. 1043-1054.

[TAN89] Tanik, M.M., y R.T. Yeh (eds.), «Rapid Prototyping in Software Development» (resultado especial), *IEEE Computer*, vol. 22, n.° 5, Mayo 1989.

[WYD96] Wyder, T., «Capturing Requirements with use-cases», *Software Development*, Febrero 1996, pp. 37-40.

[ZAH90] Zahniser, R.A., «Building Software in Groups», *American Programmer*, vol. 3, n.''' 7/8, Julio-Agosto 1990.

[ZUL92] Zultner, R., «Quality Function Deployment for Software: Satisfying Customers», *American Programrner*, Febrero 1992, pp. 28-41.

PROBLEMAS Y PUNTOS A CONSIDERAR

11.1. El análisis de requisitos del software es indudablemente la fase de comunicación más intensa del proceso de ingeniería del software. ¿Por qué suele romperse frecuentemente este enlace de comunicación?

11.2. Suele haber serias repercusiones políticas cuando comienza el análisis de requisitos del software (y/o análisis de un sistema). Por ejemplo, los trabajadores pueden creer que la seguridad de su trabajo puede verse amenazada por un nuevo sistema automático. ¿Qué origina tales problemas? ¿Se pueden llevar a cabo las tareas de análisis de manera que se minimicen estas repercusiones políticas?

11.3. Estudie su percepción ideal de la formación y currículum de un analista de sistemas.

11.4. A lo largo de todo el capítulo nos referimos al «cliente». Describa el «cliente» desde el punto de vista de los desarrolladores de sistemas de información, de los constructores de productos basados en computadora y de los

constructores de sistemas. ¡Tenga cuidado, el problema puede ser más complejo de lo que parece!

11.5. Desarrolle un «kit» de técnicas de especificación de aplicación (TFEA). El kit debería incluir un conjunto de directrices para llevar a cabo una reunión TFEA, materiales para facilitar la creación de listas y otros elementos que pudieran ayudar en la definición de requisitos.

11.6. Su profesor dividirá la clase en grupos de cuatro a seis estudiantes. La mitad del grupo hará el papel del departamento de marketing y la otra mitad el de la ingeniería del software. Su trabajo consiste en definir los requisitos del sistema de seguridad *HogarSeguro* descrito en este capítulo. Celebre una reunión TFEA usando las directrices estudiadas en el capítulo.

11.7. ¿Se puede decir que un Manual preliminar de usuario es una forma de prototipo? Razone su respuesta.

11.8. Analice el dominio de información de *HogarSeguro.* Represente (usando la notación que crea apropiada) el flujo, el contenido y cualquier estructura de la información que le parezca relevante.

11.9. Haga una partición del dominio funcional de *HogarSeguro.* Inicialmente haga una partición horizontal; después haga la partición vertical.

11.10. Construya la representación esencial y de implementación del sistema *HogarSeguro.*

11.11. Construya un prototipo en papel (o uno real) de *HogarSeguro.* Asegúrese de mostrar la interacción del propietario y el funcionamiento global del sistema.

11.12. Intente identificar componentes software de *HogarSeguro* que podrían ser «reutilizables» en otros productos o sistemas. Intente clasificar esos componentes.

11.13. Desarrolle una especificación escrita de *HogarSeguro* usando el esquema propuesto en la página web SEPA. (Nota: Su profesor le sugerirá qué secciones completar en este momento.) Asegúrese de aplicar las cuestiones descritas en la revisión de la especificación.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Los libros que indicamos sobre la ingeniería de requisitos permiten una buena base para el estudio de los conceptos y principios básicos del análisis. Thayer y Dorfman *(Software Requirements Engineering,* 2.ª ed., IEEE Computer Society Press, 1997) presentan una amplía antología sobre el tema. Graham y Graham *(Requirements Engineering and Rapid Development,* Addison-Wesley, 1989, destaca el desarrollo rápido y el uso de métodos orientados a objetos en su planteamiento sobre la ingeniería de requisitos, mientras MacCauley (*Requirements Engineering,* Springer Verlag, 1996) presenta un breve tratado académico sobre el tema.

En los últimos años, la literatura hacia énfasis en el modelo de requisitos y en los métodos de especificación, pero hoy se hace igual énfasis en los métodos eficientes para la obtención de requisitos del software. Wood y Silver *(JointApplication Development,* segunda edición, Wiley, 1995) han escrito un tratado definitivo para el desarrollo de aplicaciones enlazadas. Cohen y Cohen (*Quality Function Deployment,* Addison-Wesley, 1995), Terninko (*Step-By-Step QFD: Customer-Driven Product Design,* Saint Lucie Press, 1997), Gause y Weinberg [GAU89] y Zahniser [ZAH90] estudia los mecanismos para tener reuniones efectivas, métodos de *brainstorming* y obtención de requisitos que pueden usarse para clarificar resultados y una variedad de otras necesidades. Los casos de uso configuran una parte importante del análisis de requisitos orientado a objetos, que pueden usarse de forma independiente de la implementación tecnológica que se seleccione. Rosenburg y Scott *(Use-CaseDriven Ohject Modelling with UML: A Practical Approach,* Addison-Wesley, 1999), Schneider et al. *(Applying Use-cases: A Practica1 Guide,* Addison-Wesíey, 1998), y Texel y Williams *(Use-Cases Combined With Booch/OMT/UML,* Prentice-Hall, 1997) facilitan una guía detallada y muchos ejemplos Útiles.

El análisis del dominio de la información es un principio fundamental del análisis de requisitos. Los libros de Mattison *(The Object-Oriented Enterprise,* McGraw-Hill, 1993), y Modell *(DataAnalysis, Data Modelling and Classification,* McGraw-Hill, 1992) cubre distintos aspectos de este importante tema.

Un reciente libro de Harrison *(Prototyping and Software Development,* Springer Verlag, 1999) facilita una moderna perspectiva sobre el prototipado del software. Dos libros de Connell y Shafer *(Structured Rapid Prototyping,* Prentice-Hall, 1989) y *(Object-Oriented Rapid Prototyping,* Yourdon Press, 1994) muestra como esta importante técnica de análisis puede ser utilizada tanto para entornos convencionales como para entornos orientados a objetos.

Otros libros como los de Pomberger et al. *(Object Orientation and Prototyping in Software Engineering,* Prentice-Hall, 1996) y Krief et al. *(Prototyping With Objects,* Prentice-Hall, 1996) examina el prototipado desde la perspectiva de la orientación a objetos. La *IEEE Proceedings of International Workshop on Rapid System Prototyping* (publicado el pasado año) presenta una visión actual en esta área.

Una amplía variedad de fuentes de información sobre el análisis de requisitos y otros temas relacionados están disponibles en intemet. Una lista actualizada de páginas web que son significativas sobre los conceptos y métodos de análisis se encuentran en **http://www.pressman5.com**

CAPÍTULO 12 MODELADO DEL ANÁLISIS

EN un nivel técnico, la ingeniería del software empieza con una serie de tareas de modelado que llevan a una especificación completa de los requisitos y a una representación del diseño general del software a construir. El *modelo de análisis,* realmente un conjunto de modelos, es la primera representación técnica de un sistema. Con los años se han propuesto muchos métodos para el modelado del análisis. Sin embargo, ahora dos tendencias dominan el panorama del modelado del análisis. El primero, *análisis estructurado,* es un método de modelado clásico y se describe en este capítulo. El otro enfoque, *análisis orientado a objetos,* se estudia con detalle en el Capítulo 21. En la Sección 12.8 se presenta una breve visión general de otros métodos de análisis comúnmente usados.

El análisis estructurado es una actividad de construcción de modelos. Mediante una notación que satisfaga los principios de análisis operacional estudiada en el Capítulo 11, creamos modelos que representan el contenido y flujo de la información (datos **y** control); partimos el sistema funcionalmente, y según los distintos comportamientos establecemos la esencia de lo que se debe construir. El análisis estructurado no es un método sencillo aplicado siempre de la misma forma por todos los que lo usan. Más bien, es una amalgama que ha evolucionado durante los últimos 30 años.

En su principal libro sobre este tema, Tom DeMarco [DEM79] describe el análisis estructurado de la siguiente forma:

VISTAZO RÁPIDO

¿Qué es? **La palabra escrita es un vehículo maravilloso para la comunicación, pero no es necesariamente el mejor** camino para representar los requisitos del software. El análisis utiliza una **combinación de texto y de diagramas, para representar los requisitos de datos, funciones y comportamientos, que es relativamente fácil de entender y, más importante aún, sencillo para revisar su** corrección, **completitud** y **consistencia.**

¿Quién lo **hace?** El ingeniero del software **(a veces llamado analista) construye el modelo utilizando los requisitos definidos por el cliente.**

¿Por qué es importante? **Para validar los requisitos del software necesitamos examinarlos** desde diferentes puntos de vista. El análisis representa los requisitos en tres «dimensiones», por esa razón, **se** incrementa la probabilidad de encontrar errores, descubrir inconsistencias y detectar omisiones.

¿C son los pasos? Los requisitos de datos, funciones y **comportamientos son** modelados utilizando dife**rentes** diagramas. **El modelado de datos define objetos de** datos, **atributos y relaciones. El modelado de** funciones indica **como los** datos **son** transformados dentro **del** sistema. **El** modelado del comportamiento representa el impacto de los sucesos. Se crean unos modelos preliminares que **son** analizados y refinados para valorar **su** claridad. completitud y consistencia. Una **especificación incorporada en el modelo** es creada **y luego validada, tanto por** el ingeniero **del software, como por los** clientes/usuarios.

¿Cuál es el producto **obtenido? Las descripciones de los objetos de datos, los diagramas entidad-**relación, los **diagramas de flujo de datos, los diagra**mas de transición de estados, las especificaciones del proceso y las especificaciones de control son creadas como resultado de **las** acti**vidades** del **análisis.**

¿Cómo puedo estar **seguro de que** lo **he hecho correctamente?** El resultado del modelado del análisis debe ser revisado para verificar **su** corrección, completitud y consistencia.

Observando **los** problemas y fallos reconocidos para la fase de análisis, se puede sugerir que necesitamos añadir los siguientes objetivos a la fase de análisis.

- Los productos del análisis han de ser de mantenimiento muy fácil. Esto concierne concretamente al documento final [Especificación de Requisitos del Software].
- Se deben tratar **los** problemas de gran tamaño mediante algún método efectivo de partición. La especificación mediante novelas victorianas ya no sirve.
- Siempre que sea posible, se deben utilizar gráficos.
- Tenemos que diferenciar las consideraciones lógicas [esenciales] y las físicas [de implementación]...

Como mínimo, necesitamos ...

- Algo que nos ayude a hacer una partición de **los** requisitos y a documentar esas divisiones antes de especificar ...
- Algún método de seguimiento y evaluación de interfaces ...
- Nuevas herramientas para describir la lógica y la táctica, algo mejores que descripciones narrativas ...

Es muy probable que ningún otro método de ingeniería del software haya generado tanto interés, haya sido probado (y a veces rechazado **y** vuelto a probar) por tanta gente, criticado y haya provocado tanta controversia. Pero el método ha subsistido y ha alcanzado un importante seguimiento dentro de la comunidad de la ingeniería del software.

12.1 BREVE HISTORIA

Al igual que muchas de las contribuciones importantes a la ingeniería del software, el análisis estructurado no fue introducido en un solo artículo o libro clave que incluyera un tratamiento completo del tema. Los primeros trabajos sobre modelos de análisis aparecieron a finales de los 60 y principios de los 70, pero la primera aparición del enfoque de análisis estructurado fue como complemento de otro tema importante —el «diseño estructurado»—. **Los** investigadores (por ejemplo, [STE74], [YOU78], necesitaban una notación gráfica para representar los datos y los procesos que los transforman. Esos procesos quedarían finalmente establecidos en una arquitectura de diseño.

Cita:
El problema no es que existan problemas. El problema es esperar que vengan y pensa que el tener problemas es un problema.
Theodore Rubin

El término «análisis estructurado» originalmente acuñado por Douglas Ross, fue popularizado por DeMarco [DEM79]. En su libro sobre esta materia, DeMarco presentó y denominó los símbolos gráficos y los modelos que los incorporan. En los años siguientes, Page-Jones [PAG80], Gane y Sarson [GAN82] y muchos otros propusieron variaciones del enfoque del análisis estructurado. En todos los casos, el método se centraba en aplicaciones de sistemas de información y no proporcionaba una notación adecuada para los aspectos de control y de comportamiento de los problemas de ingeniería de tiempo real.

A mediados de los 80, las «ampliaciones» para tiempo real fueron introducidas por Ward y Mellor [WAR85] y, más tarde, por Hatley y Pirbhai [HAT87]. Con esas ampliaciones, se consiguió un método de análisis más robusto que podía ser aplicado de forma efectiva a problemas de ingeniería. En la actualidad, se está intentando desarrollar una notación consistente [BRU88] y se están publicando tratamientos modernos que permitan acomodar el uso de herramientas CASE [YOU89].

12.2. LOS ELEMENTOS DEL MODELO DE ANÁLISIS

El modelo de análisis debe lograr tres objetivos primarios: (1) describir lo que requiere el cliente, (2) establecer una base para la creación de un diseño de software, y **(3)** definir un conjunto de requisitos que se pueda validar una vez que se construye el software. Para lograr estos objetivos, el modelo de análisis extraído durante el análisis estructurado toma la forma ilustrada en la Figura 12.1.

En el centro del modelo se encuentra el *diccionario de datos* —un almacén que contiene definiciones de todos los objetos de datos consumidos y producidos por el software—. Tres diagramas diferentes rodean el núcleo. El *diagrama de entidad-relación (DER)* representa las relaciones entre los objetos de datos. El DER es la notación que se usa para realizar la actividad de modelado de datos. Los atributos de cada objeto de datos señalados en el DER se puede describir mediante una descripción de objetos de datos.

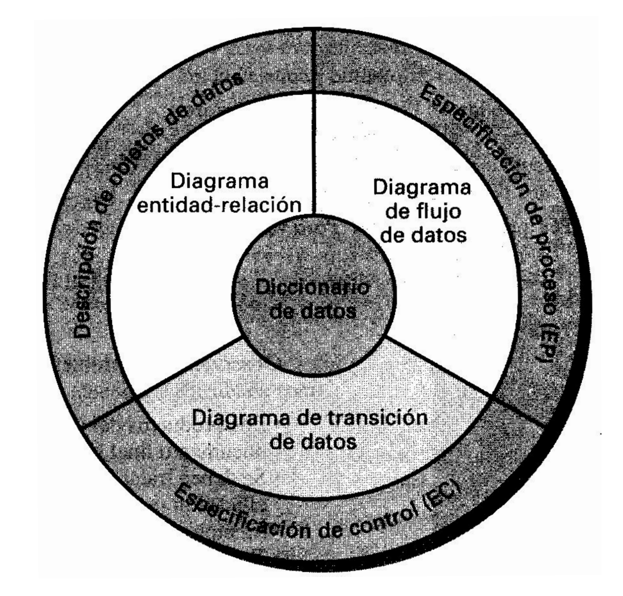


FIGURA 12.1. La estructura del modelo de análisis.

El *diagrama de flujo de datos (DFD)* sirve para dos propósitos: (1) proporcionar una indicación de cómo se transforman los datos a medida que se avanza en el sistema, y (2) representar las funciones (y subfunciones) que transforman el flujo de datos. El DFD proporciona información adicional que se usa

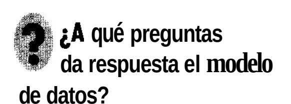
durante el análisis del dominio de información y sirve como base para el modelado de función. En una *especificación de proceso (EP)* se encuentra una descripción de cada función presentada en el DFD.

El *diagrama de transición de estados (DTE)* indica cómo se comporta el sistema como consecuencia de sucesos externos. Para lograr esto, el DTE representa los diferentes modos de comportamiento (llamados *estados)* del sistema y la manera en que se hacen las transiciones de estado a estado. El DTE sirve como la base del modelado de comportamiento. Dentro de la *especificación de control (EC)* se encuentra más información sobre los aspectos de control del software.

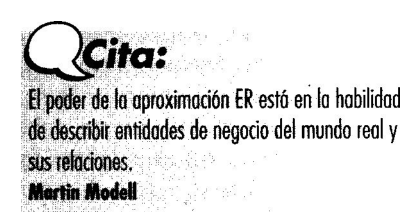
El modelo de análisis acompaña a cada diagrama, especificación y descripción, y al diccionario señalado en la Figura 12.1. Un estudio más detallado de estos elementos del modelo de análisis se presenta en las secciones siguientes.

12.3. MODELADO DE DATOS

El modelado de datos responde a una serie de preguntas específicas importantes para cualquier aplicación de procesamiento de datos. ¿Cuáles son los *objetos de datos* primarios que va a procesar el sistema? ¿Cuál es la composición de cada objeto de datos y qué atributos describe el objeto? ¿Dónde residen actualmente los objetos? ¿Cuál es la relación entre los objetos y los procesos que los transforman?



Para responder estas preguntas, los métodos de modelado de datos hacen uso del diagrama de entidad-relación (DER). El DER, descrito con detalle posteriormente en esta sección, permite que un ingeniero del software identifique objetos de datos y sus relaciones mediante una notación gráfica. En el contexto del análisis estructurado, el DER define todos los datos que se introducen, se almacenan, se transforman y se producen dentro de una aplicación.



El diagrama entidad-relación se centra solo en los datos (y por consiguiente satisface el primer principio operacional de análisis), representando una «red de datos» que existe para un sistema dado. El DER es específicamente Útil para aplicaciones en donde los datos y las relaciones que gobiernan los datos son complejos. A diferencia del diagrama de flujo de datos (estudiado en la Sección 12.4 y usado para representar como se transforman los datos), el modelado de datos estudia los datos independientemente del procesamiento que los transforma.

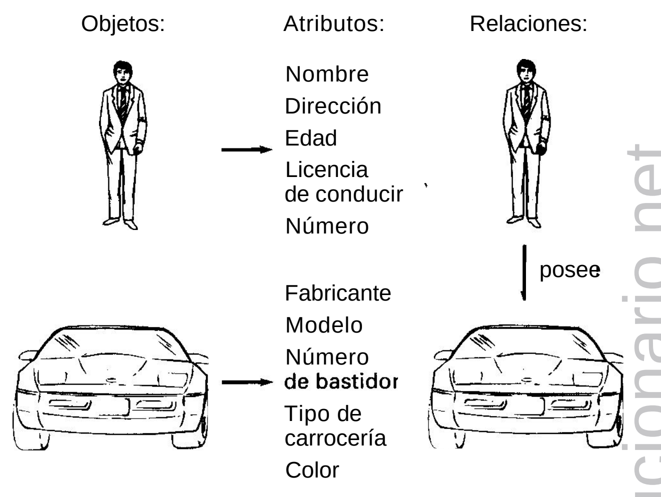


FIGURA 12.2. Objetos de datos, atributos y relaciones.

12.3.1. Objetos de datos, atributos y relaciones

El modelo de datos se compone de tres piezas de información interrelacionadas: el objeto de datos, los atributos que describen el objeto de datos y la relación que conecta objetos de datos entre sí.

Objetos de datos. Un *objeto de datos* es una representación de cualquier composición de información compuesta que deba comprender el software. Por composición de información, entendemos todo aquello que tiene un número de propiedades o atributos diferentes. Por tanto, el «ancho» (un valor sencillo) no sería un objeto de datos válido, pero las «dimensiones» (incorporando altura, ancho y profundidad) se podría definir como objeto.

CLAVE

Un objeto de datos es una representación de cualquier configuración de información que es procesada por el software.

Un objeto de datos puede ser una entidad externa (por ejemplo, cualquier cosa que produce o consume información), una cosa (por ejemplo, un informe o una pantalla), una ocurrencia (por ejemplo, una llamada telefónica) o suceso (por ejemplo, una alarma), un puesto

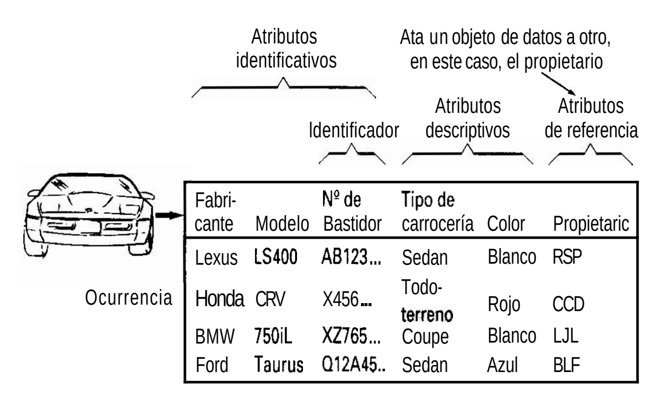
(por ejemplo, un vendedor), una unidad de la organización (por ejemplo, departamento de contabilidad), o una estructura (por ejemplo, un archivo). Por ejemplo, una persona o un coche (Fig. 12.2) se pueden ver como un objeto de datos en el sentido en que cualquiera se puede definir según un conjunto de atributos. La *descripción de objeto de datos* incorpora el objeto de datos y todos **sus** atributos.



Información útil sobre el modelo de datos puede encontrarse en **www.datamodel.org**

Los objetos de datos se relacionan con otros. Por ejemplo, **persona** puede *poseer* **coche,** donde la relación *poseer* implica una «conexión» específica entre **persona y coche.** Las relaciones siempre se definen por el contexto del problema que se está analizando.

Un objeto de datos solamente encapsula datos —no hay referencia dentro de un objeto de datos a operaciones que actúan en el dato'—. Por consiguiente, el objeto de datos se puede representar como una tabla de la forma en que se muestra en la Figura **12.3.** Los encabezamientos de la tabla reflejan atributos del objeto. En este caso, se define un coche en términos de fabricante, modelo, número de bastidor, tipo de carrocería, color y propietario. El cuerpo de la tabla representa ocurrencias del objeto de datos. Por ejemplo, un Honda CRV es una ocurrencia del objeto de datos **coche.**



Fabricante	Modelo	Nº de Bastidor	Tipo de carrocería	Color	Propietario
Lexus	LS400	AB123...	Sedan	Blanco	RSP
Honda	CRV	X456...	Todo-terreno	Rojo	CCD
BMW	750iL	XZ765...	Coupe	Blanco	LJL
Ford	Taurus	Q12A45..	Sedan	Azul	BLF

FIGURA 12.3. Representación tabular del objeto de datos.

Atributos. Los atributos definen las propiedades de un objeto de datos y toman una de las tres características diferentes. Se pueden usar para (1) nombrar una ocurrencia del objeto de datos, (2) describir la ocurrencia, o *(3)* hacer referencias a otra ocurrencia en otra tabla. Además, uno o varios atributos se definen como un *identificador* -es decir, el atributo identificador supone una «clave» cuando queramos encontrar una instancia del objeto de dato—. En algunos casos, los valores para los identificadores son únicos, aunque esto no es un requisito. Haciendo referencia al objeto de datos **coche,** un identificador razonable podría ser el número de bastidor.



los atributos definen un objeto de datos, describen sus caracteristicos, y en algunas ocasiones, establecen referencias o otros objetos.

El conjunto de atributos apropiado para un objeto de datos dado se determina mediante el entendimiento del contexto del problema. Los atributos para **coche** descritos anteriormente podrían servir para una aplicación que usara un Departamento de Vehículos de Motor, pero estos atributos serían menos útiles para una compañía de automóviles que necesite fabricar software de control. En este último caso, los atributos de **coche** podrían incluir también número de bastidor, tipo de carrocería, y color, pero muchos atributos adicionales (Por ejemplo: código interior, tipo de dirección, selector de equipamiento interior, tipo de transmisión) se tendrían que añadir para que **coche** sea un objeto significativo en el contexto del control de fabricación.



las relaciones indican la manero en que los objetos de datos están «conectados» entre sí.

Relaciones. Los objetos de datos se conectan entre sí de muchas formas diferentes. Considere dos objetos de datos, **libro** y **librería.** Estos objetos se pueden representar mediante la notación simple señalada en la Figura 12.4a. Se establece una conexión entre **libro** y **librería** porque los dos objetos se relacionan. Pero, ¿qué son relaciones? Para determinar la respuesta, debemos comprender el papel de libro y librería dentro del contexto del software que se va construir. Podemos definir un conjunto de parejas objeto-relación que definen las relaciones relevantes. Por ejemplo,

- una librería pide libros
- una librería muestra libros
- una librería almacena libros
- una librería vende libros
- una librería devuelve libros

[1] Esta distinción separa el objeto de datos de la *clase* u *objeto* definidos como parte del paradigma orientado a objetos que se estudia en la Parte Cuarta de este libro.

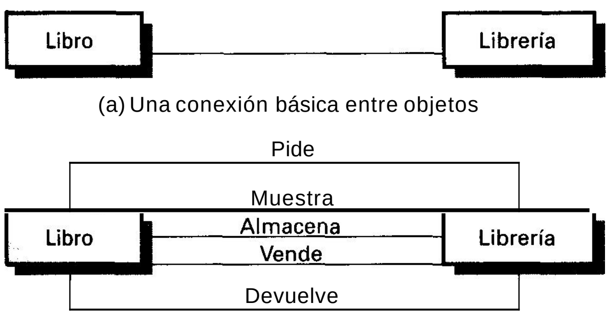


FIGURA 12.4. Relaciones.

Las relaciones *pide, muestra, almacena* y *devuelve* define las conexiones relevantes entre **libro** y **librería.** La Figura 12.4b ilustra estas parejas objeto-relación gráficamente.

Es importante destacar que las parejas objeto-relación tienen dos direcciones; esto es, se pueden leer en cualquier dirección. Una librería *pide* libros o los libros *son pedidos por* una librería[2].

12.3.2. Cardinalidad y modalidad

Los elementos básicos del modelado de datos - objetos de datos, atributos, y relaciones — proporcionan la base del entendimiento del dominio de información de un problema. Sin embargo, también se debe comprender la información adicional relacionada con estos elementos básicos.

Hemos definido un conjunto de objetos y representado las parejas objeto-relación que los limitan. Una simple referencia: el **objeto X** que *se relaciona con* el **objeto Y** no proporciona información suficiente para propósitos de ingeniería del software. Debemos comprender la cantidad de ocurrencias del **objeto X** que están relacionadas con ocurrencias del **objeto Y.** Esto nos conduce al concepto del modelado de datos llamado cardinalidad.

Cardinalidad. El modelo de datos debe ser capaz de representar el número de ocurrencias de objetos que se dan en una relación. Tillman [TIL93] define la *cardinalidad* de una pareja objeto-relación de la forma siguiente:

La cardinalidad es la especificación del número de ocurrencias [de un objeto] que se relaciona con ocurrencias de otro [objeto]. La cardinalidad normalmente se expresa simplemente como «uno» o «muchos». Por ejemplo, un marido puede tener solo *una* esposa (en la mayoría de las culturas), mientras que un padre puede tener *muchos* hijos. Teniendo en consideración todas las combinaciones de «uno» y «muchos», dos [objetos] se pueden relacionar como:

- Uno a uno (1:1)—Una ocurrencia [de un objeto] «A» se puede relacionar a una y sólo una ocurrencia del objeto «B», y una ocurrencia de «B» se puede relacionar sólo con una ocurrencia de «A».
- Uno a muchos (1:N)—Una ocurrencia del objeto «A» se puede relacionar a una o muchas ocurrencias del objeto «B», pero una de «B» se puede relacionar sólo a una ocurrencia de «A». Por ejemplo, una madre puede tener muchos hijos, pero un hijo sólo puede tener una madre.
- Muchos a muchos (M:N)—Una ocurrencia del objeto «A» puede relacionarse con una o más ocurrencias de «B», mientras que una de «B» se puede relacionar con una o más de «A». Por ejemplo, un tío puede tener muchos sobrinos, mientras que un sobrino puede tener muchos tíos.

La cardinalidad define «el número máximo de relaciones de objetos que pueden participar en una relación» [TIL93]. Sin embargo, no proporciona una indicación de si un objeto de datos en particular debe o no participar en la relación. Para especificar esta información, el modelo de datos añade modalidad a la pareja objeto-relación.

Modalidad. La *modalidad* de una relación es cero si no hay una necesidad explícita de que ocurra una relación, o que sea opcional. La modalidad es 1 **si** una ocurrencia de la relación es obligatoria. Para ilustrarlo, consideremos el software que utiliza una compañía telefónica local para procesar peticiones de reparación. Un cliente indica que hay un problema. Si se diagnostica que un problema es relativamente simple, sólo ocurre una única acción de reparación. Sin embargo, si el problema es complejo, se pueden requerir múltiples acciones de reparación. La Figura 12.5 ilustra la relación, cardinalidad y modalidad entre los objetos de datos **cliente** y **acción de reparación.**

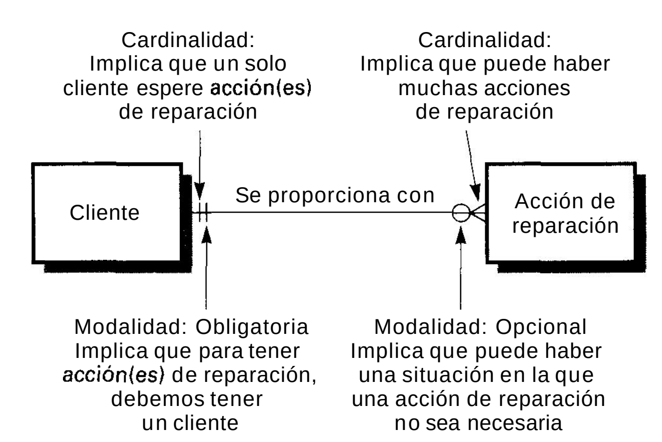
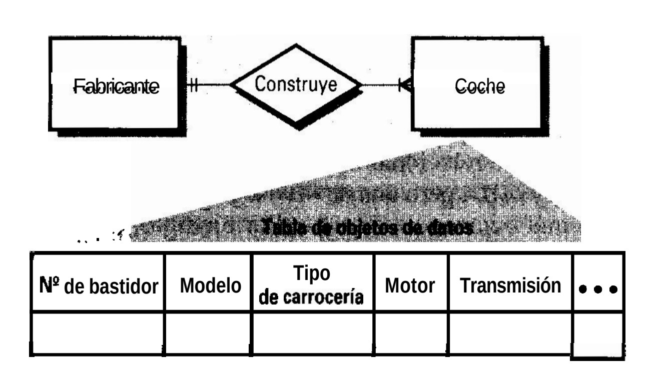


FIGURA 12.5. Cardinalidad y modalidad.

[2] Para evitar cualquier ambiguedad, se debe considerar la manera en que se etiqueta una relación. Por ejemplo, si no se considera el contexto para una relación bidireccional, la Figura 12 4b podría interpretarse erróneamente como libros piden librerías. En tales casos, se necesita volver a escribir la frase.



Nº de bastidor	Modelo	Tipo de carrocería	Motor	Transmisión	•••

FIGURA 12.6. Un DER y una tabla de objetos de datos simples (Nota: En este DER la relación «construye» se indica mediante un rombo sobre la línea de conexión entre los objetos de datos).

En la figura, se establece una relación de cardinalidad de 1 a muchos. Esto es, se puede proporcionar un sólo cliente con cero o muchas acciones de reparación. Los símbolos sobre la conexión de relación que está más cerca de los rectángulos del objeto de datos indica cardinalidad. La barra vertical indica **1, y** la que está dividida en tres indica muchas. La modalidad se indica por los símbolos más lejanos de los rectángulos de objetos de datos. La segunda barra vertical a la izquierda indica que debe haber un cliente para que ocurra una acción de reparación. El círculo de la derecha indica que puede no existir acción de reparación para el tipo de problema informado por el usuario.

12.3.3. Diagramas Entidad-Relación

La pareja objeto-relación (estudiada en la Sección 12.3.1) es la piedra angular del modelo de datos. Estas parejas se pueden representar gráficamente mediante el *diagrama entidad relación (DER).* El DER fue propuesto originalmente por Peter Chen [CHE77] para el diseño de sistemas de bases de datos relacionales **y** ha sido ampliado por otros. Se identifica un conjunto de componentes primarios para el DER: objetos de datos, atributos, relaciones y varios indicadores tipo. El propósito primario del DER es representar objetos de datos y sus relaciones.



El objetivo principal de un DER es representar entidades (objetos de datas) y sus relaciones entre sí.

Una notación DER rudimentaria se ha presentado en la Sección 12.3. Los objetos de datos son representados por un rectángulo etiquetado. Las relaciones se indican mediante una línea etiquetada conectando objetos. En algunas variaciones del DER, la línea de conexión contiene un rombo que se etiqueta con la relación. Las *conexiones* entre objetos de datos y relaciones se establecen mediante una variedad de símbolos especiales que indican cardinalidad y modalidad (Sección 12.3.2).



Un detallado estudio que describe los diagramas entidad relación puede encontrarse en **www.univ-paris1.fr/CRINFO/dmrg/MEE/misop003/**

La relación entre los objetos de datos **coche y fabricante** se representarían como se muestra en la Figura 12.6. Un fabricante construye uno o muchos coches. Dado el contexto implicado por el DER, la especificación del objeto de datos **coche** (consulte la tabla de objetos de datos de la Figura 12.6) sería radicalmente diferente de la primera especificación (Fig. 12.3) Examinando los símbolos al final de la línea de conexión entre objetos, se puede ver que la modalidad de ambas incidencias es obligatoria (las líneas verticales).

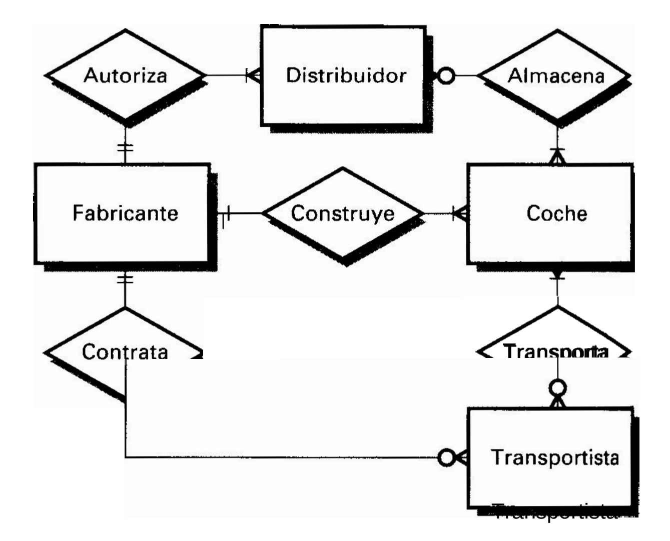


FIGURA 12.7. DER ampliado.

Ampliando el modelo, representamos un DER extremadamente simplificado (Figura 12.7) del elemento de distribución del negocio de los automóviles. Se han introducido nuevos objetos de datos, **transportista** y **distribuidor.** Además, las relaciones nuevas *—transporta, contrata, autoriza* y *almacena*— indican cómo los objetos de datos de la figura se asocian entre sí. Las tablas para cada objeto de datos del DER, se tendría que desarrollar de acuerdo con las reglas presentadas al comienzo del capítulo.

Además de la notación de DER básica introducida en las Figuras 12.6 **y** 12.7, el analista puede representar *jerarquías de tipos de objetos de datos.* En muchas ocurrencias, un objeto de datos realmente puede representar una clase o categoría de información. Por ejemplo, el objeto de datos **coche** puede categorizarse como doméstico, europeo o asiático. La notación del DER mostrada en la Figura 12.8 representa esta categorización en forma de jerarquía.



Desarrolle el DER de forma iterativa poro ir refinando los objetos de datas y los relaciones que los conectan.

La notación del DER también proporciona un mecanismo que representa la asociabilidad entre objetos. Un *objeto de datos asociativo* se representa como se muestra en la Figura 12.9. En la figura, los objetos de datos que modelan los subsistemas individuales se asocian con el objeto de datos **coche.**

El modelado de datos y el diagrama entidad relación proporcionan al analista una notación concisa para examinar datos dentro del contexto de una aplicación de procesamiento de datos. En la mayoría de los casos, el enfoque del modelado de datos se utiliza para crear una parte del modelo de análisis, pero también se puede utilizar para el diseño de bases de datos **y** para soportar cualquier otro método de análisis de requisitos.

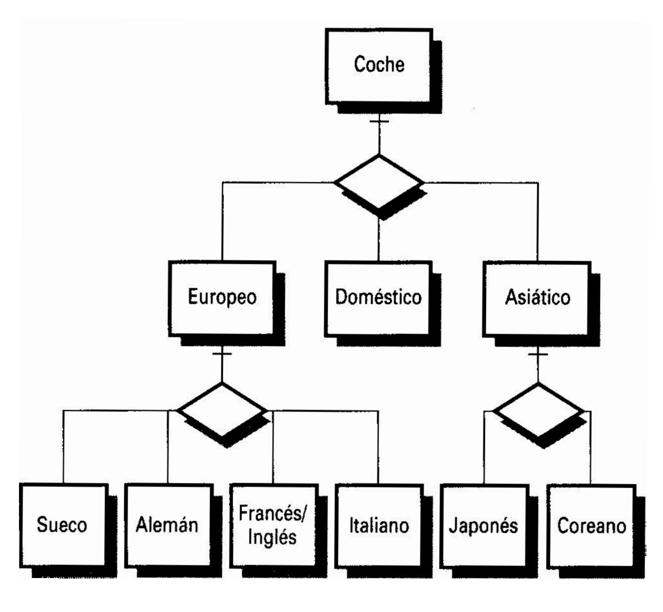


FIGURA 12.8. Jerarquía tipo de objetos de datos.

12.4 MODELADO FUNCIONAL Y FLUJO DE INFORMACIÓN

La información se transforma a medida que *fluye* por un sistema basado en computadora. El sistema acepta entradas en una gran variedad de formas; aplica elementos de hardware, software y humanos para transformar la entrada en salida, y produce salida en una gran variedad de formas. La entrada puede ser una señal de control transmitida por un controlador, una serie de números escritos por un enlace de una red o un archivo voluminoso de datos recuperado de un almacenamiento secundario. La transformación puede ser, desde una sencilla comparación lógica, hasta un complejo algoritmo numérico o un mecanismo de reglas de inferencia de un sistema experto. La salida puede ser el encendido de un diodo de emisión de luz (LED) o un informe de 200 páginas. Efectivamente, podemos crear un *modelo de flujo* para cualquier sistema de computadora, independientemente del tamaño y de la complejidad.

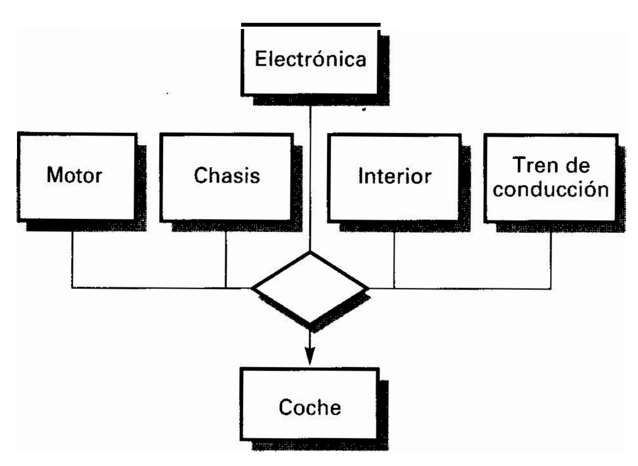


FIGURA 12.9. Asociación de objetos de datos.

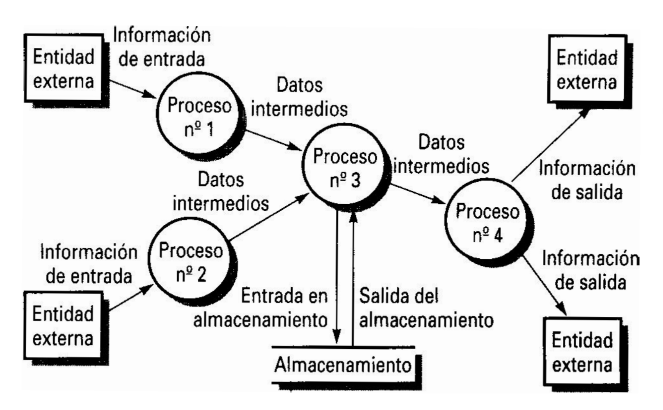


FIGURA 12.10. Modelo del flujo de información.

El análisis estructurado es una técnica del modelado del flujo **y** del contenido de la información. Tal como muestra la Figura 12.10, el sistema basado en computadora se representa como una transformación de información. Se utiliza un rectángulo para representar una *entidad externa,* esto es, un elemento del sistema (por ejemplo, un elemento hardware, una persona, otro programa) u otro sistema que produce información para ser transformada por el software, o recibe información producida por el software. Un círculo (también llamado burbuja) representa un *proceso* o *transformación* que es aplicado a los datos (o al control) y los modifica. Una flecha representa uno o más *elementos de datos* (objetos de dato). Toda flecha en un diagrama de flujos de datos debe estar etiquetada. Las líneas en paralelo representan un almacenamiento —información almacenada que es utilizada por el software—. La simplicidad de la notación del DFD es una razón por la que las técnicas del análisis estructurado son ampliamente utilizadas.



*El DFD no es procedimental. **No** permite representar con su notación **gráfica** tratamientos **condicionales** ni bucles. Simplemente muestra el **flujo** de datos.*

Es conveniente hacer notar que el diagrama no proporciona ninguna indicación explícita de la secuencia de procesamiento o de una condición lógica. El procedimiento o secuencia puede estar implícitamente en el diagrama, pues los detalles lógicos son generalmente retrasados hasta el diseño del software. Es importante no confundir un DFD con el diagrama de flujo.

12.4.1. Diagramas de flujo de datos

A medida que la información se mueve a través del software, es modificada por una serie de transformaciones. El *diagrama de flujo de datos (DFD)* es una técnica que representa el flujo de la información y las transformaciones que se aplican a los datos al moverse desde la entrada hasta la salida. En la Figura 12.10 se muestra la forma básica de un diagrama de flujo de datos. El DFD es también conocido como *grafo de flujo de datos o* como *diagrama de burbujas.*



CLAVE

*El DFD facilita un mecanismo para modelizar el **flujo** de información y modelizar las funciones.*

Se puede usar el diagrama de flujo de datos para representar un sistema o un software a cualquier nivel de abstracción. De hecho, los DFDs pueden ser divididos en niveles que representen un mayor flujo de información y un mayor detalle funcional. Por consiguiente, el DFD proporciona un mecanismo para el modelado funcional, así como el modelado del flujo de información. Al hacer esto, se cumple el segundo principio de análisis operacional (esto es, crear un modelo funcional) estudiado en el Capítulo 11.

Un DFD de nivel 0, también denominado *modelo fundamental del sistema o modelo de contexto,* representa al elemento de software completo como una sola burbuja con datos de entrada y de salida representados por flechas de entrada y de salida, respectivamente. Al dividir el DFD de nivel 0 para mostrar más detalles, aparecen representados procesos (burbujas) y caminos de flujo de información adicionales. Por ejemplo, un DFD de nivel 1 puede contener cinco o seis burbujas con flechas interconectadas. Cada uno de los procesos representados en el nivel 1 es una subfunción del sistema general en el modelo del contexto.



*La descomposición de un nivel de DFD en **su** siguiente debería seguir una aproximación al **ratio** 1:5, reduciendo las futuras descomposiciones*

Como ya indicamos anteriormente, se puede refinar cada una de las burbujas en distintos niveles para mostrar un mayor detalle. La Figura 12.11 ilustra este concepto. El modelo fundamental del sistema *F* muestra la entrada principal *A* y la salida final *B*. Refinamos el modelo *F* en las transformaciones *f1 a f7*. Observe que se debe mantener la *continuidad del flujo de información,* es decir, que la entrada y la salida de cada refinamiento debe ser la misma. Este concepto, a veces denominado *balanceo,* es esencial para el desarrollo de modelos consistentes. Un mayor refinamiento de *f4* muestra más detalle en la forma de las transformaciones *f41 a f45*. De nuevo, la entrada (X, Y) y la salida (Z) permanecen inalteradas.

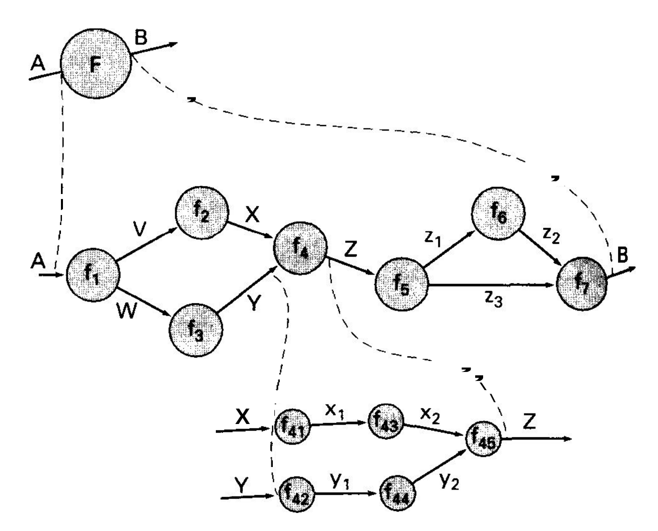


FIGURA 12.11. *Refinamiento del flujo de información.*

La notación básica que se usa para desarrollar un **DFD** no es en **sí** misma suficiente para describir los requisitos del software. Por ejemplo, una flecha de un DFD representa un objeto de datos que entra o sale de un proceso. Un almacén de datos representa alguna colección organizada de datos. Pero ¿cuál es el *contenido* de los datos implicados en las flechas o en el almacén? Si la flecha (o el almacén) representan una colección de objetos, ¿cuáles son? Para responder a estas preguntas, aplicamos otro componente de la notación básica del análisis estructurado —el *diccionario de datos*—. El formato y el uso del diccionario de datos se explica más adelante.



*Si bien la continuidad del flujo de información debe ser mantenido, reconocemos que un elemento de **datos** representado en un nivel puede ser **refinado** en sus partes constituyentes en el siguiente nivel.*

La notación gráfica debe ser ampliada con texto descriptivo. Se puede usar una *especificación de proceso* (EP) para especificar los detalles de procesamiento que implica una burbuja del DFD. La especificación de proceso describe la entrada a la función, el algoritmo que

se aplica para transformar la entrada y la salida que se produce. Además, el EP indica las restricciones y limitaciones impuestas al proceso (función), las características de rendimiento que son relevantes al proceso y las restricciones de diseño que puedan tener influencia en la forma de implementar el proceso.

12.4.2. Ampliaciones para sistemas de tiempo real

Muchas aplicaciones de software son dependientes del tiempo y procesan más información orientada al control de datos. Un sistema de tiempo real debe interactuar con el mundo real en marcos temporales que vienen dados por el mundo real. El control de naves o de procesos de fabricación, los productos de consumo y la instrumentación industrial son algunas de entre cientos de aplicaciones del software de tiempo real.

Para que resulte adecuado el análisis del software de tiempo real, se han propuesto varias ampliaciones para la notación básica del análisis estructurado.

12.4.3. Ampliaciones de Ward y Mellor

Ward y Mellor [WAR85] amplían la notación básica del análisis estructurado para que se adapte a las siguientes demandas impuestas por los sistemas de tiempo real:

- Flujo de información que es recogido o producido de forma continua en el tiempo.
- Información de control que pasa por el sistema y el procesamiento de control asociado.
- Ocurrencias múltiples de la misma transformación que se encuentran a menudo en situaciones de multitarea.
- Estados del sistema y mecanismos que producen transición de estados en el sistema.

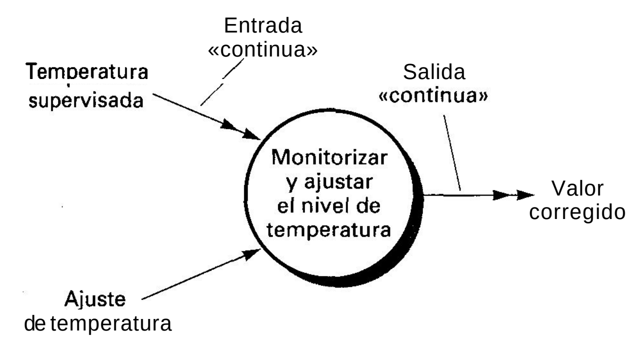


FIGURA 12.12. Flujo de datos continuo en el tiempo.

En un porcentaje significativo de aplicaciones de tiempo real, el sistema debe controlar la información *continua en el tiempo* generada por algún proceso del mundo real. Por ejemplo, puede que se haga necesario un sistema de control de pruebas en tiempo real para máquinas de turbina de gas, para que se supervise la velocidad de la turbina, la temperatura del combustible y varias sondas de presión, todo ello de forma continua. La notación del flujo de datos convencional no hace distinciones entre datos discretos y datos continuos en el tiempo. Una ampliación de la notación básica del análisis estructurado que se muestra en la Figura 12.12 proporciona un mecanismo para representar el flujo de datos continuo en el tiempo. Para representar el flujo continuo en el tiempo se usa la flecha de dos cabezas, mientras que el flujo de datos discreto se representa con una flecha de una sola cabeza. En la figura, se mide continuamente la **temperatura supervisada,** mientras que sólo se proporciona un valor para el **ajuste de temperatura.** El proceso de la figura produce una salida continua, **valor corregido.**



Para odecuor el modelo a un sistema en tiempo real, la notación del análisis estructurado debe permitir procesar eventos y la llegada de continuos datos.

La distinción entre flujo de datos discreto y continuo en el tiempo tiene implicaciones importantes, tanto para el ingeniero de sistemas como para el diseñador del software. Durante la creación del modelo del sistema, podrá aislar los procesos que pueden ser críticos para el rendimiento (es muy usual que la entrada y salida de datos continuos en el tiempo dependan del rendimiento). Al crear el modelo físico o de implementación, el diseñador debe establecer un mecanismo para la recolección de datos continuos en el tiempo. Obviamente, el sistema digital recolecta datos en una forma casi continua utilizando técnicas de muestreo de alta velocidad. La notación indica dónde se necesita hardware de conversión analógica a digital y qué transformaciones requerirán, con mayor probabilidad, un software de alto rendimiento.

En los diagramas de flujo de datos convencionales no se representa explícitamente el control o *flujos de sucesos.* De hecho, al analista se le advierte de que ha de excluir específicamente la representación del flujo de control del diagrama de flujo de datos. Esta exclusión es demasiado restrictiva cuando se trata de aplicaciones de tiempo real y, por este motivo, se ha desarrollado una notación especializada para la representación del flujo de sucesos y del procesamiento de control. Siguiendo con los convenios establecidos para los diagramas de flujo de datos, el flujo de datos se representa mediante flechas con trazo continuo. Sin embargo, ***el** flujo de control* se representa mediante flechas de trazo discontinuo o sombreadas. Un proceso que sólo maneja flujos de control denominado *proceso de control,* se representa analógicamente mediante una burbuja con trazo discontinuo.



FIGURA 12.13. Flujos de datos y de control utilizando la notación de Ward y Mellor [WAR85].

El flujo de control puede ser una entrada directa de un proceso convencional o de un proceso de control. La Figura 12.13 ilustra el flujo de control y su procesamiento, utilizando la notación de Ward y Mellor. La figura ilustra el planteamiento de mayor nivel de un flujo de datos y de control de una célula de fabricación. A medida que se van colocando en las fijaciones los componentes a ser ensamblados por un robot, se ajusta un bit de un **buffer de estado de las piezas** (un almacén de control) que indica la presencia o ausencia de cada componente. La información de sucesos contenida en el **buffer de estado de las piezas** se pasa como una **cadena de bits** a un proceso, *interfaz del monitor de fijación y operador*. El proceso leerá **Órdenes del operador** sólo cuando la información de control, cadena de bits, indique que todas las fijaciones contienen componentes. Se envía un indicador de suceso, **indicador de comienzo/parada**, a *control de activación del robot,* un proceso de control que permite un posterior procesamiento de órdenes. Como consecuencia del suceso **activar proceso** se producen otros flujos de datos que son enviados a *procesar órdenes del robot.*

Cita:
El entorno de un sistema en tiempo real a veces contiene dispositivos que actúan como los 5 sentidos del sistema.
Paul Ward y Stephen Mellor

En algunas situaciones, en un sistema de tiempo real puede haber ocurrencias múltiples del mismo proceso de control o de transformación de datos. Se puede dar este caso en un entorno multitarea en el que se activan las tareas como resultado de algún procesamiento interno o de sucesos externos.

Por ejemplo, puede que haya que supervisar varios buffers de estado de piezas para que puedan activarse diferentes robots en los momentos oportunos. Además, puede que cada robot tenga su propio sistema de control. La notación de Ward y Mellor se usa para representar *múltiples ocurrencias equivalentes* del mismo proceso.

12.4.4. Ampliaciones de Hatley y Pirbhai

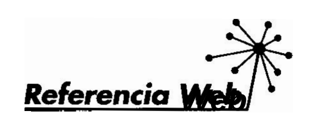
Las ampliaciones de Hatley y Pirbhai [HAT87] a la notación básica del análisis estructurado se centra menos en la creación de símbolos gráficos adicionales y más en la representación y especificación de los aspectos del software orientados al control. La flecha de trazo discontinuo se utiliza para representar el flujo de control o de sucesos. A diferencia de Ward y Mellor, Hatley y Pirbhai sugieren que se represente por separado la notación de trazo continuo de la de trazo discontinuo. Así, se define un *diagrama de flujo de control (DFC)*. El DFC contiene los mismos procesos que el DFD, pero muestra el flujo de control en lugar de datos. En lugar de representar directamente los procesos de control dentro del modelo de flujo, se usa una referencia de notación (una barra sólida) a una *especificación de control (EC)*. En esencia, se puede considerar la barra como una «ventana» hacia un «director» (la EC) que controla los procesos (las burbujas) representadas en el DFD, de acuerdo con los sucesos que pasan a través de la ventana. Se usa la EC, que se describe en detalle en la Sección 12.6.4, para indicar: (1) cómo se comporta el software cuando se detecta un suceso o una señal de control, y **(2)** qué procesos se invocan como consecuencia de la ocurrencia del suceso. Se usa una *especificación de proceso (EP)* para describir el procesamiento interno de los procesos representados en el diagrama de flujo.



CLAVE

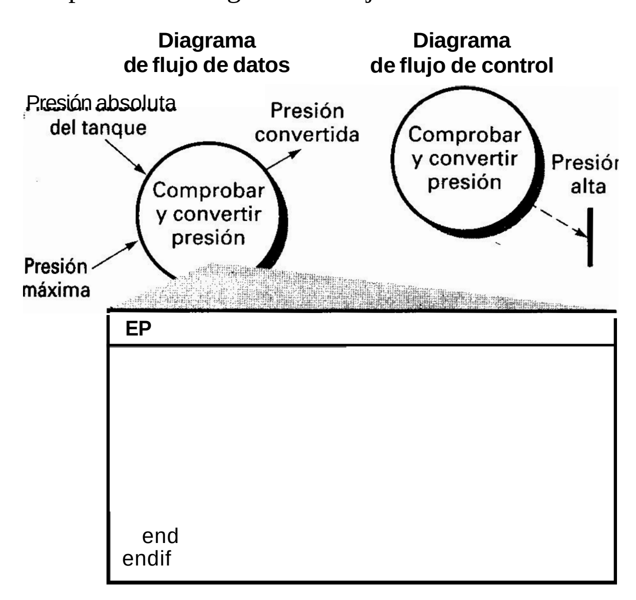
El **DFC** muestra como los eventos se mueven en el sistema. Una EC muestra el comportamiento del software como una consecuencia de los eventos y a qué procesos les corresponde intervenir en la manipulación de los eventos.

Usando la notación descrita en las Figuras 12.12 y 12.13, junto con la información adicional contenida en EPs y ECs, Hatley y Pirbhai crean un modelo de sistema de tiempo real. Para representar los datos y los procesos que los manejan se usan los diagramas de flujo de datos. Los diagramas de flujo de control muestran cómo fluyen los sucesos entre los distintos procesos e ilustran cómo los sucesos externos hacen que se activen los procesos. Las interrelaciones entre los modelos de proceso de control se muestran esquemáticamente en la Figura 12.14. El modelo de procesamiento está «conectado» con el modelo de control a través de *condiciones de datos.* El modelo de control está «conectado» con el modelo de procesos mediante la información de activación de procesos contenida en la EC.



La especificación del sistema en tiempo real planteada por Hatley y Pirbhai es descrita en **www.univ-paris1.fr/CRINFO/dmrg/MEE98/misop032/**

Una condición de datos se produce cuando los datos de entrada de un proceso hacen que se produzca una salida de control. La Figura 12.14 ilustra esta situación en una parte del modelo de flujo para un sistema de supervisión y control automatizado de válvulas de presión de una refinería de petróleo. El *proceso comprobar* y *convertir presión* implementa el algoritmo descrito en el pseudocódigo EP que se muestra. Cuando la presión absoluta del tanque es mayor que el máximo permitido, se genera un suceso **presión alta.** Observe que cuando se usa la notación de Hatley y Pirbhai, el flujo de datos de muestra como parte de un DFD, mientras que el flujo de control se encuentra a parte en un diagrama de flujo de control.



Para determinar qué es lo que ocurre cuando se produce este suceso, debemos comprobar la EC.

La especificación de control (EC) contiene varias herramientas importantes del modelado. Para indicar qué procesos se activan por un suceso dado que fluye por la barra vertical, se usa una *tabla de activación de procesos* (descrita en la Sección 12.6.4). Por ejemplo, una tabla de activación de procesos (TAP) para la Figura 12.14 podría indicar que el suceso **presión alta** haría que se invocara un proceso *reducir presión del tanque* (que no se muestra). Además de la TAP, la EC puede contener un *diagrama de transición de estados (DTE).* El DTE es un modelo de comportamiento que se basa en la definición de un *conjunto de estados del sistema* **y** se describe en la sección siguiente.

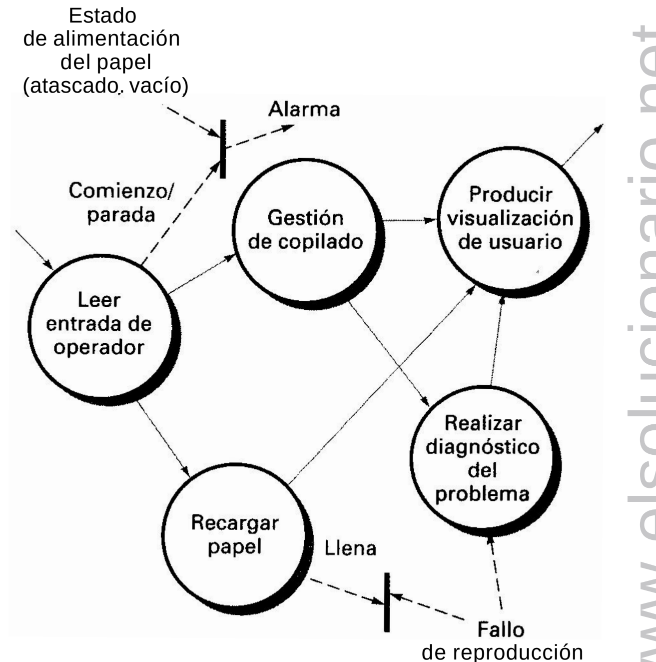
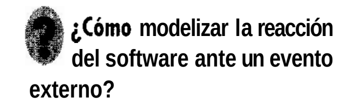


FIGURA 12.15. DFC de nivel 1 para el software de una fotocopiadora.

12.5 MODELADO DEL COMPORTAMIENTO

El *modelado del comportamiento* es uno de los principios fundamentales de todos los métodos de análisis de requisitos. Sin embargo, sólo algunas versiones ampliadas del análisis estructurado ([WAR85], [HAT87]) proporcionan una notación para este tipo de modelado. El diagrama de transición de estados representa el comportamiento 'de un sistema que muestra los estados y los sucesos que hacen que el sistema cambie de estado. Además, el DTE indica **qué** acciones (por ejemplo, activación de procesos) se llevan a cabo como consecuencia de un suceso determinado.



Un *estado* es un modo observable de comportamiento. Por ejemplo, estados para un sistema de supervisión **y** de control para que las válvulas de presión descritas en la Sección 12.4.4. puedan estar en: *estado de supervisión, estado de alarma, estado de liberación depresión* **y** otros. Cada uno de estos estados representa un modo de comportamiento del sistema. Un diagrama de transición de estados indica cómo se mueve el sistema de un estado a otro.

Para ilustrar el uso de las ampliaciones de comportamiento y de control de Hatley y Pirbhai, consideremos el software empotrado en una máquina fotocopiadora de oficina. En la Figura 12.15 se muestra un flujo de control para el software de fotocopiadora. Las flechas del flujo de datos se han sombreado ligeramente con propósitos ilustrativos, pero en realidad se muestran como parte de un diagrama de flujo de control.

Los flujos de control se muestran de cada proceso individual y las barras verticales representan las «ventanas» EC. Por ejemplo, los sucesos **estado de alimentación del papel** y de **comienzo/parada** fluyen dentro de la barra de EC. Esto implica que cada uno de estos sucesos hará que se active algún proceso representado en el DFC. Si se fueran a examinar las interioridades del EC, se mostraría el suceso **comienzo/parada** para activar/desactivar el proceso de *gestión de copiado.* De forma similar, el suceso **atascada** (parte del estado de alimentación del papel) activaría *realizar diagnóstico del problema.* Se debería destacar que todas las barras verticales dentro del DFC se refieren a la misma EC. Un flujo de suceso se puede introducir directamente en el proceso como muestra **fallo de reproducción.** Sin embargo, este flujo no activa el proceso, sino que proporciona información de control para el algoritmo de proceso.

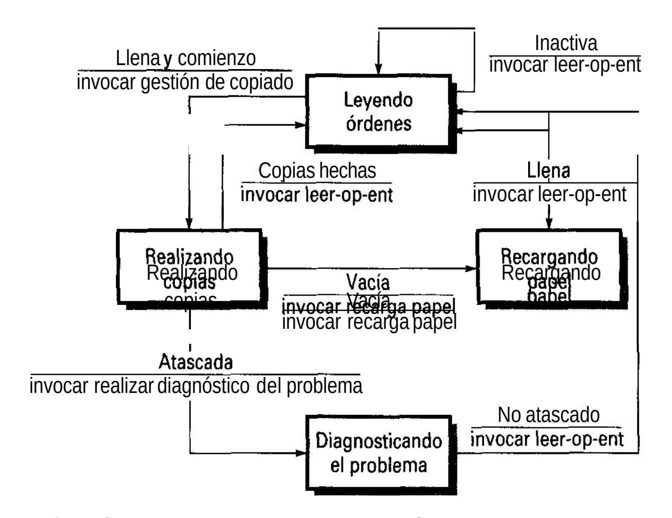
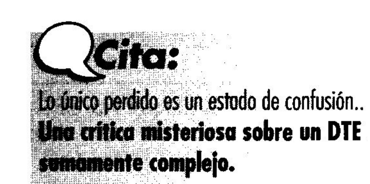


FIGURA 12.16. Diagramas de transición de estados simplificado para **el software** de una fotocopiadora.

Un diagrama de transición de estados simplificado para el software de fotocopiadora descrito anteriormente se muestra en la Figura 12.16. Los rectángulos representan estados del sistema y las flechas representan *transiciones* entre estados. Cada flecha está etiquetada con una expresión en forma de fracción. La parte superior indica el suceso (o sucesos) que hace(n) que se produzca la transición. La parte inferior indica la acción que se produce como consecuencia del suceso. Así, cuando la bandeja de papel está llena, **y** el botón de comienzo ha sido pulsado, el sistema pasa del estado *leyendo órdenes* al estado *realizando copias.* Observe que los estados no se corresponden necesariamente con los procesos de forma biunívoca. Por ejemplo, el estado *realizando copias* englobaría tanto el proceso *gestión de copiado* como el proceso *producir visualización de usuario* que aparecen en la Figura 12.15.



12.6 MECANISMOS DEL ANÁLISIS ESTRUCTURADO

En la sección anterior, hemos visto las notaciones básica y ampliada del análisis estructurado. Para poder utilizarlas eficientemente en el análisis de requisitos del software, se ha de combinar esa notación con un conjunto de heurísticas que permitan al ingeniero del software derivar un buen modelo de análisis. Para ilustrar el uso de esas heurísticas, en el resto de este capítulo utilizaremos una versión adaptada de las ampliaciones de Hatley y Pirbhai [HAT87] a la notación básica del análisis estructurado.



El modelo de análisis permite un exámen crítico de los requisitos del software desde tres puntos diferentes de vista. Asegurando la utilidad de los DERs, DFDs y DTEs cuando construyes el modelo

En las secciones siguientes, se examina cada uno de los pasos que **se** debe seguir para desarrollar modelos completos y precisos mediante el análisis estructurado. A lo largo de este estudio, se usará la notación presentada en la Sección 12.4 y se presentarán, con algún detalle, otras formas de notación ya aludidas anteriormente.

12.6.1. Creación de un diagrama Entidad-Relación

El diagrama de entidad-relación permite que un ingeniero del software especifique los objetos de datos que entran y salen de un sistema, los atributos que definen las propiedades de estos objetos y las relaciones entre los objetos. Al igual que la mayoría de los elementos del modelo de análisis y las relaciones entre objetos, el DER se construye de una forma iterativa. Se toma el enfoque siguiente:



1. Durante la recopilación de requisitos, se pide que los clientes listen las «cosas» que afronta el proceso de la aplicación o del negocio. Estas «cosas» evolucionan en una lista de objetos de datos de entrada y de salida, así como entidades externas que producen o consumen información.
2. Tomando objetos uno cada vez, el analista y el cliente definen si existe una conexión (sin nombrar en ese punto) o no entre el objeto de datos y otros objetos.
3. Siempre que existe una conexión, el analista y el cliente crean una o varias parejas de objeto-relación.
4. Para cada pareja objeto-relación se explora la cardinalidad y la modalidad.
5. Interactivamente se continúan los pasos del 2 al 4 hasta que se hayan definido todas las parejas objeto-relación. Es normal descubrir omisiones a medida que el proceso continúa. Objetos y relaciones nuevos se añadirán invariablemente a medida que crezca el número de interacciones.
6. Se definen los atributos de cada entidad.
7. Se formaliza y se revisa el diagrama entidad-relación.
8. Se repiten los pasos del 1 al 7 hasta que se termina el modelado de datos.

Para ilustrar el uso de estas directrices básicas, usaremos el ejemplo del sistema de seguridad *HogarSeguro* tratado en el Capítulo 11. A continuación, reproducimos la narrativa de procesamiento para *HogarSeguro* (Sección 11.3.3), la siguiente lista (parcial) de «cosas» son importantes para el problema:

- propietario
- panel de control
- sensores
- sistema de seguridad
- servicio de vigilancia

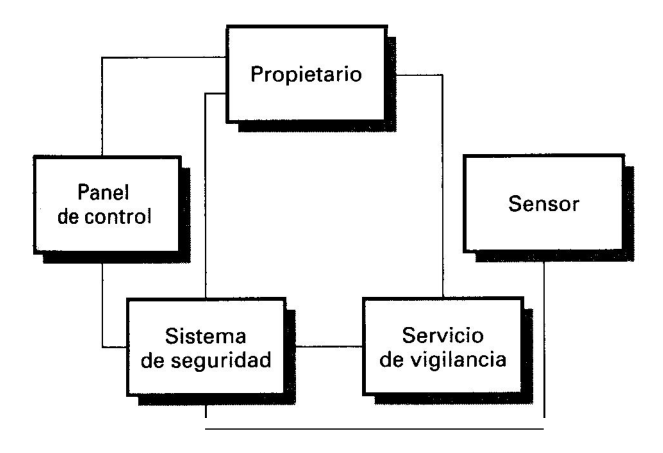


FIGURA 12.17. Establecimiento de conexiones.

Tomando estas «cosas» una a una, se exploran las conexiones. Para realizar esto, se dibuja cada objeto y se señalan las líneas que conectan objetos. Por ejemplo, la Figura 12.17 muestra que existe una conexión directa entre el **propietario** y **panel de control, sistema de seguridad** y **servicio de vigilancia.** Existe una conexión Única entre el sensor y el sistema de seguridad, y así sucesivamente.

Una vez que se han definido todas las conexiones, se identifican una o varias parejas objeto-relación para cada conexión. Por ejemplo, se determina la conexión entre **sensor** y **sistema de seguridad** para que tenga las parejas objeto-relación siguientes:

el **sistema de seguridad** *supervisa* el **sensor**
el **sistema de seguridad** *activaldesactiva* el **sensor**
el **sistema de seguridad** *prueba* el **sensor**
el **sistema de seguridad** *programa* el **sensor**

Cada una de las parejas objeto-relación anteriores se analizan para determinar la cardinalidad y la modalidad. Por ejemplo, en la pareja objeto-relación el **sistema de seguridad** *supervisa* el **sensor,** la cardinalidad entre **sistema de seguridad** y **sensor** es una a muchos. La modalidad es una incidencia de **sistema de seguridad** (obligatoria) y al menos una incidencia de **sensor** (obligatorio). Mediante la notación DER introducida en la Sección 12.3, la línea de conexión entre sistema de seguridad y sensor se modificaría, como se muestra en la Figura 12.18. Se aplicaría un análisis similar al resto de los objetos de datos.

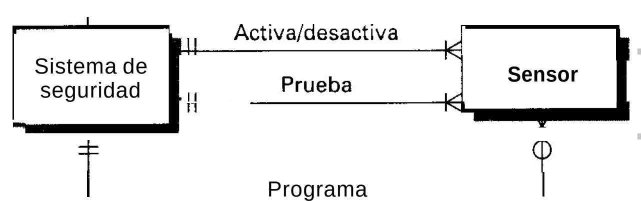


FIGURA 12.18. Desarrollo de relaciones y cardinalidad/ modalidad.

Se estudia cada objeto para determinar sus atributos. Como se está considerando el software que debe soportar *HogarSeguro*, los atributos se deberían centrar en datos que deban almacenarse para permitir que opere el sistema. Por ejemplo, el objeto **sensor** podría tener los atributos siguientes: tipo de sensor, número de identificación interna, localización de zona, y nivel de alarma.

12.6.2. Creación de un modelo de flujo de datos

El *diagrama de flujo de datos (DFD)* permite al ingeniero del software desarrollar los modelos del ámbito de información y del ámbito funcional al mismo tiempo. A medida que se refina el DFD en mayores niveles de detalle, el analista lleva a cabo implícitamente una descomposición funcional del sistema. Al mismo tiempo, el refinamiento del DFD produce un refinamiento de los datos a medida que se mueven a través de los procesos que componen la aplicación.



Unas pocas directrices sencillas pueden ayudar de forma considerable durante la derivación de un diagrama de flujo de datos; (1) el diagrama de flujo de datos de nivel 0 debe reflejar el software/sistema como una sola burbuja; (2) se deben anotar cuidadosamente la entrada y la salida principales; *(3)* el refinamiento debe comenzar aislando los procesos, los objetos de datos y los almacenes de datos que sean candidatos a ser representados en el siguiente nivel; (4) todas las flechas y las burbujas deben ser rotuladas con nombres significativos; *(5)* entre sucesivos niveles se debe mantener la *continuidad del flujo de información; (6)* se deben refinar las burbujas de una en una. Hay una tendencia natural a complicar en exceso el diagrama de flujo de datos. Esto ocurre cuando el analista intenta reflejar demasiados detalles demasiado pronto o representa aspectos procedimentales en detrimento del flujo de información.

En la Figura 12.19 se muestra el DFD de nivel 0 para *HogarSeguro.* Las entidades externas principales (cuadros) producen información para ser usada por el sistema y consumen información generada por el sistema. Las flechas etiquetadas representan objetos de datos u objetos de datos de tipo jerárquico. Por ejemplo, **órdenes y datos de usuario** engloba todas las órdenes de configuración, todas las órdenes de activación/desactivación, todas las variadas interacciones y todos los datos que se introducen para cualificar o ampliar una orden.

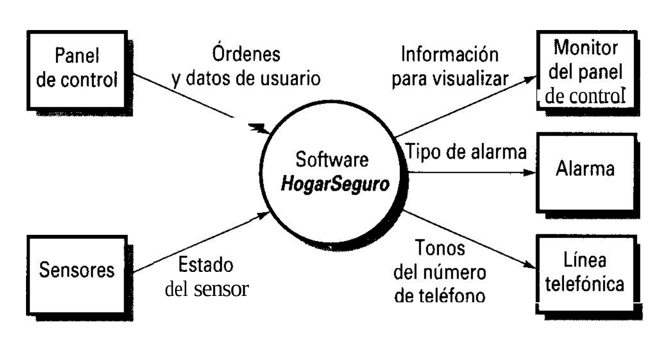


FIGURA 12.19. DFD de nivel contextual para *HogarSeguro.*

Ahora tenemos que expandir el DFD de nivel 0 o de nivel a un modelo de nivel 1. Pero, ¿cómo lo hacemos? Una sencilla, pero efectiva, técnica consiste en realizar un «análisis gramatical» de la narrativa de procesamiento que describe la burbuja de nivel contextual. Es decir, aislar todos los nombres (y sentencias nominales) y todos los verbos (y sentencias verbales) de la narrativa presentada arriba. Para ilustrarlo, reproducimos de nuevo la narrativa de procesamiento, subrayando las primeras ocurrencias de los nombres y con las primeras ocurrencias de los verbos en cursiva. (Se debería señalar que se omiten los nombres y los verbos que son sinónimos y no se apoyan directamente en el proceso de modelado)[3].



El análisis gramatical no es seguro, pero facilito una excelente plataforma si estás empezando a definir objetas de datos y su transformación.

El software HogarSeguro *permite* al propietario de la vivienda *configurar* el sistema de seguridad al *instalarlo; supervisa* todos **los** sensores *conectados* al sistema de seguridad e *interactúa* con el propietario a través de un teclado numérico y unas teclas de función que *se encuentran* en el panel de control de *HogarSeguro* que se muestra en la Figura 11.2.

Durante la instalación, se usa el panel de control de HogarSeguro para *«programar»* y *configurar* el sistema. Cada **sensor** tiene *asignado* un número y un tipo, existe una contraseña maestra para *activar* y *desactivar* el sistema, y se *introduce(n)* un(os) teléfono(s) con **los** que *contactar* cuando se produce un suceso detectado por un sensor.

Cuando el software *detecta* un suceso, *invoca* una alarma audible que está incorporada en el sistema. Tras un retardo, *especificado* por el propietario durante la configuración del sistema, el programa marca un número de teléfono de un servicio de monitorización, *proporciona* información sobre la situación e *informa* sobre la naturaleza del suceso detectado. Cada 20 segundos *se volverá a marcar* el número de teléfono hasta que se consiga *establecer* la comunicación.

Toda la interacción con HogarSeguro está *gestionada* por un subsistema de interacción con el usuario que *lee* la información introducida a través del teclado numérico y de las teclas de función, muestra mensaies de petición en un monitor LCD y *muestra* información sobre el estado del sistema en el monitor LCD. La interacción **por** teclado toma la siguiente forma...

De acuerdo con el «análisis gramatical», comienza a aparecer un patrón. Todos los verbos son procesos de *HogarSeguro,* es decir, deben estar en última instancia representados como burbujas en el consiguiente DFD. Todos los nombres son, o bien entidades externas (cuadrados), objetos de datos o de control (flechas) o almacenes de datos (líneas dobles). Además los nombres y los verbos están relacionados entre sí (por ejemplo, cada sensor tiene *asignado* un número y un tipo). Por tanto, con el análisis gramatical de la narrativa de procesamiento de una burbuja de cualquier nivel de DFD, podemos generar mucha información útil sobre cómo proceder en el refinamiento del siguiente nivel. Usando esa información, obtenemos el DFD de nivel 1 que se muestra en la Figura 12.20. El proceso de nivel contextual de la Figura 12.19 se ha expandido en siete procesos, derivados de un examen del análisis gramatical. De forma similar se ha derivado el flujo de información entre los procesos del nivel 1.

[3] Se debe tener en cuenta que serán omitidos los nombres y verbos que sean sinónimos o no tengan una relación directa con el modelo de procesos.

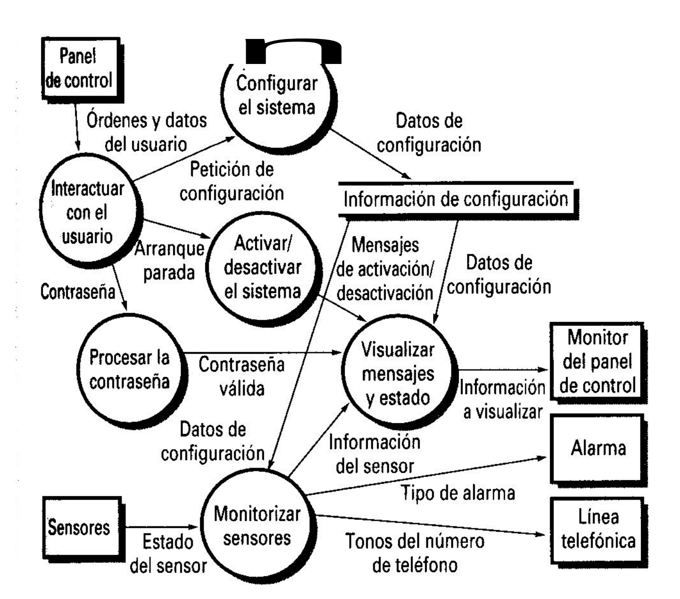


FIGURA 12.20. DFD de nivel 1 para *HogarSeguro.*

Se debe tener en cuenta que entre los niveles 0 y 1 se ha mantenido la continuidad del flujo de información. Se pospone hasta la Sección **12.7** la elaboración de los contenidos de las entradas y las salidas de los DFD de niveles 0 y 1.



Es cierto que el proceso narrativo pretende analizar lo escrito siempre con el mismo nivel de abstracción.

Se pueden refinar en posteriores niveles los procesos representados en el DFD de nivel 1. Por ejemplo, se puede refinar el proceso *monitorizar sensores* al DFD de nivel **2** que se muestra en la Figura **12.21.** Podemos observar de nuevo que se mantiene la continuidad del flujo de información entre los niveles.

El refinamiento de los DFDs continúa hasta que cada burbuja representa una función sencilla.

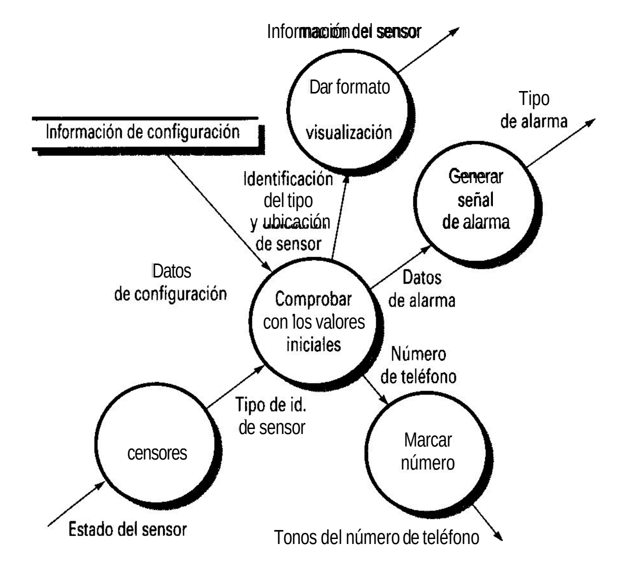


FIGURA 12.21. DFD de nivel **2** que refina el proceso *monitorizar sensores.*

12.6.3. Creación de un modelo de flujo de control

Para muchos tipos de aplicaciones de procesamiento de datos, todo lo que se necesita para obtener una representación significativa de los requisitos del software es el modelo de flujo de datos. Sin embargo, como ya hemos mencionado anteriormente, existe una clase de numerosas aplicaciones que están «dirigidas» por sucesos en lugar de por los datos, que producen información de control más que informes o visualizaciones, que procesan información con fuertes limitaciones de tiempo y rendimiento. Tales aplicaciones requieren un modelado del flujo de control además del modelado del flujo de información.

La notación gráfica que se requiere para crear un *diagrama de flujo de control (DFC)* ya ha sido presentada en la Sección **12.4.4.** Recordando la forma en que se crea un DFC, lo primero es «eliminar» del modelo de flujo de datos todas las flechas de flujo de datos. Luego, se añaden al diagrama los sucesos y los elementos de control (flechas con línea discontinua), así como «ventanas» (barras verticales) a las especificaciones de control. Pero, ¿cómo se seleccionan los sucesos?

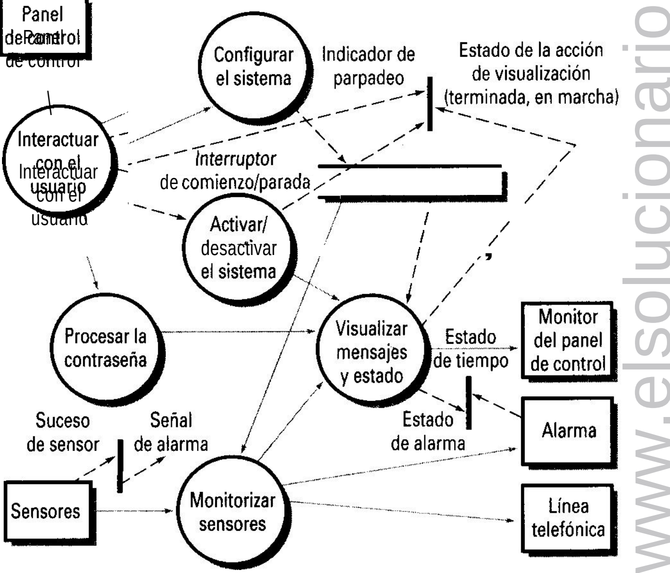


FIGURA 12.22. DFD de nivel I para *HogarSeguro.*

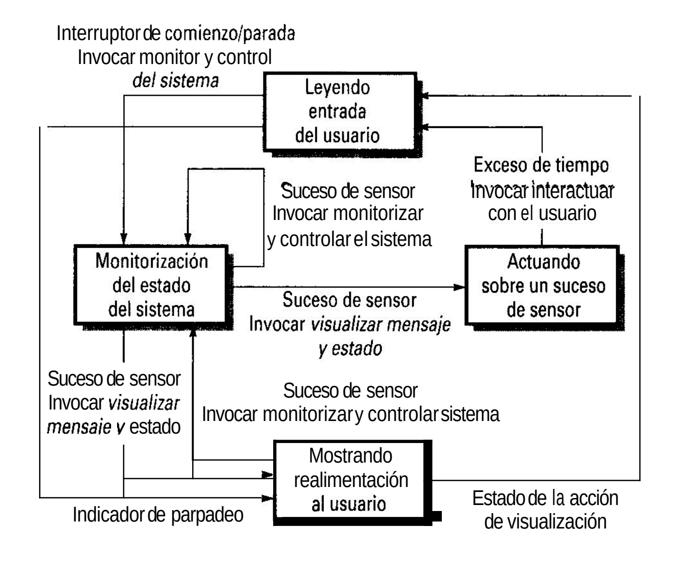
Ya hemos señalado que los sucesos o los elementos de control se implementan como valores lógicos (por ejemplo, *verdadero* o *falso,* si *o no, 1* ó *0)* o como una lista discreta de condiciones (*vacía, atascada, llena*). Para seleccionar posibles candidatos a sucesos, se pueden sugerir las siguientes directrices:

- Listar todos los sensores que son «leídos» por el software.
- Listar todas las condiciones de interrupción.
- Listar todos los «interruptores» que son accionados por el operador.
- Listar todas las condiciones de datos.
- De acuerdo con el análisis de nombres y verbos que se aplicó a la narrativa de procesamiento, revisar todos los «elementos de control» como posibles entradas/salidas de ECs.

- Describir el comportamiento del sistema identificando sus estados; identificar cómo se alcanza cada estado y definir las transiciones entre los estados.
- Centrarse en las posibles omisiones —un error muy común en la especificación del control (por ejemplo, preguntarse si existe alguna otra forma en la que se puede llegar a un estado o salir de él)—.



En la Figura **12.22** se ilustra un DFC de nivel 1 para el software *HogarSeguro.* Entre los sucesos y los elementos de control que aparecen están **suceso de sensor** (por ejemplo, un sensor ha detectado una anomalía), **indicador de parpadeo** (una señal para que el monitor LCD parpadee), e **interruptor de comienzo/parada** (una señal para encender y apagar el sistema). Un suceso fluye a una ventana EC desde el mundo exterior, ello implica la activación por la EC de uno o más procesos de los que aparecen en el DFC. Cuando un elemento de control emana de un proceso y fluye a una ventana EC, ello implica el control y la activación de algún proceso o de una entidad externa.



12.6.4. La especificación de control

La especificación de control (EC) representa el comportamiento del sistema (al nivel al que se ha hecho referencia) de dos formas diferentes. La EC contiene un *diagrama de transición de estados (DTE)* que es una *especificación secuencia1* del comportamiento. También puede contener una *tabla de activación de procesos (TAP)* —una *especificación combinatoria* del comportamiento—. En la Sección **12.4.4.** se presentaron los atributos inherentes de la EC. Ahora es el momento de examinar un ejemplo de esta importante notación del modelado del análisis estructurado.

La Figura **12.23** refleja un *diagrama de transición de estados* para el modelo de flujo de nivel 1 de *HogarSeguro.* Las flechas de transición etiquetadas indican cómo responde el sistema a los sucesos a medida que pasa por los cuatro estados definidos en este nivel. Estudiando el DTE, un ingeniero del software puede determinar el comportamiento del sistema y, **lo** que es más importante, comprobar si hay «lagunas» en el comportamiento especificado.

Una representación algo diferente del modo de comportamiento es la *tabla de activación de procesos (TAP).* La TAP representa la información contenida en el DTE dentro del contexto de los procesos, no de los estados. Es decir, la tabla indica los procesos (burbujas) del modelo de flujo que serán invocados cuando se produzca un suceso. Se puede usar la TAP como una guía para el diseñador que tenga que construir un controlador que controle los procesos representados en ese nivel. En la Figura **12.24** se muestra una TAP para el modelo de flujo de nivel 1 de *HogarSeguro.*

La EC describe el comportamiento del sistema, pero no nos proporciona información sobre el funcionamiento interno de los procesos que son activados como resultado de ese comportamiento. En la siguiente sección se estudia la notación del modelado que proporciona esa información.

12.6.5. La especificación del proceso

Se usa la especificación de proceso (EP) para describir todos los procesos del modelo de flujo que aparecen en el nivel final de refinamiento. El contenido de la especificación de procesamiento puede incluir una narrativa textual, una descripción en *lenguaje de diseño de programas (LDP)* del algoritmo del proceso, ecuaciones matemáticas, tablas, diagramas o gráficos. Al proporcionar una EP que acompañe cada burbuja del modelo de flujo, el ingeniero del software crea una «mini-especificación» que sirve como primer paso para la creación de la *Especificación de Requisitos del Software* y constituye una guía para el diseño de la componente de programa que implementará el proceso.

Sucesos de entrada						
Suceso de sensor	0	0	0	0	1	0
Indicador de parpadeo	0	0	1	1	0	0
Interruptor de comienzo/parada	0	1	0	0	0	0
Estado de la acción de visualización						
Terminada	0	0	0	1	0	0
En marcha	0	0	1	0	0	1
Exceso de tiempo	0	0	0	0	0	1
Salida						
Señal de alarma	0	0	0	0	1	0
Activación de procesos						
Monitorizar y controlar el sistema	0	1	0	0	1	1
Activar/desactivar el sistema	0	1	0	0	0	0
Visualizar mensajes y estado	1	0	1	1	1	1
Interactuar con el usuario	1	0	0	1	0	1

FIGURA 12.24. Tabla de activación de procesos para *HogarSeguro.*

Para ilustrar el uso de la EP consideremos el proceso que representa la transformación del modelo de flujo de *Hogarseguro* (Figura 12.20). La EP para esta función puede tomar la siguiente forma:

EP: proceso

Procesar la contraseña lleva a cabo **la** validación de todas las contraseñas del sistema *HogarSeguro*. Procesar la contraseña recibe una contraseña de 4 dígitos desde la función interactuar con el usuario. La contraseña primero se compara con la contraseña maestra almacenada en el sistema. Si al contrastar con la contraseña maestra, <contraseña validada = true> se pasa **a** la función visualizar mensajes **y** estado. Si la comparación no *es* correcta, los cuatro dígitos se comparan con una lista secundaria de contraseñas (estas pueden asignarse a invitados o trabajadores que necesitan entrar en la casa cuando el propietario no está presente). Si la contraseña coincide con alguna de las de la lista, <contraseña validada = true> es pasada a la función visualizar mensajes y estado. Si no existe coincidencia, <contraseña validada = false> se pasa a la función visualizar mensajes **y** estado.

Si en esta etapa se desea incluir detalles algorítmicos adicionales, se puede incluir como parte de la EP su representación en lenguaje de descripción de programa. Sin embargo, muchos piensan que se debe posponer la versión en LDP hasta que comience el diseño.

12.7 EL DICCIONARIO DE DATOS

El modelo de análisis acompaña representaciones de objetos de datos, funciones y control. En cada representación los objetos de datos y/o elementos de control juegan un papel importante. Por consiguiente, es necesario proporcionar un enfoque organizado para representar las características de cada objeto de datos y elemento de control. Esto se realiza con el diccionario de datos.

Se ha propuesto el diccionario *de* datos como gramática casi formal para describir el contenido de los objetos definidos durante el análisis estructurado. Esta importante notación de modelado ha sido definida de la siguiente forma [YOU89]:

> El diccionario de datos es un listado organizado de todos los elementos de datos que son pertinentes para el sistema, con definiciones precisas **y** rigurosas que permiten que el usuario y el analista del sistema tengan una misma comprensión de las entradas, salidas, de las componentes de los almacenes **y** [también] de los cálculos intermedios.

¿Cómo representar el contenido de los objetos de datos que han sido identificados?

Actualmente, casi siempre se implementa el diccionario de datos como parte de una «herramienta CASE de análisis y diseño estructurados». Aunque el formato del diccionario varía entre las distintas herramientas, la mayoría contiene la siguiente información:

- *nombre*—el nombre principal del elemento de datos o de control, del almacén de datos, o de una entidad externa.
- alias — otros nombres usados para la primera entrada.
- dónde se *usa/cómo se* usa—un listado de los procesos que usan el elemento de datos o de control y cómo lo usan (por ejemplo, como entrada al proceso, como salida del proceso, como almacén de datos, como entidad externa).
- Descripción *del contenido*—el contenido representado mediante una notación.
- *Información* adicional-otra información sobre los tipos de datos, los valores implícitos (si se conocen), las restricciones o limitaciones, etc.

Una vez que se introducen en el diccionario de datos un nombre y sus alias, se debe revisar la consistencia de las denominaciones. Es decir, si un equipo de análisis decide denominar un elemento de datos recién derivado como **xyz,** pero en el diccionario ya existe otro llamado **xyz,** la herramienta CASE que soporta el diccionario muestra un mensaje de alerta sobre la duplicidad de nombres. Esto mejora la consistencia del modelo de análisis y ayuda a reducir errores.

La información de «dónde se usa/cómo se usa» se registra automáticamente a partir de los modelos de flujo. Cuando se crea una entrada del diccionario, la herramienta CASE inspecciona los DFD y los DFC para determinar los procesos que usan el dato o la información de control y cómo lo usan. Aunque esto pueda no parecer importante, realmente es una de las mayores ventajas del diccionario. Durante el análisis, hay una corriente casi continua de cambios. Para proyectos grandes, a menudo es bastante difícil determinar el impacto de un cambio. Algunas de las preguntas que se plantea el ingeniero del software son «¿dónde se usa este elemento de datos? ¿qué mas hay que cambiar si lo modificamos? ¿cuál será el impacto general del cambio?». **Al** poder tratar el diccionario de datos como una base de datos, el analista puede hacer preguntas basadas en «dónde se usa/cómo se usan y obtener respuestas a peticiones similares a las anteriores.



Herramientas **CASE**
Análisis Estructurado

La notación utilizada para desarrollar una descripción de contenido se indica en la siguiente tabla:

Construcción de datos	Notación	Significado
Agregación	–	está compuesto de
Secuencia	+	Y
Selección	[|]	uno u otro
Repetición	{ 1"	n repeticiones de
	O	datos opcionales
	...	delimitadores de comentarios

La notación permite al ingeniero del software representar una composición de datos en una de las tres alternativas fundamentales que pueden ser construidas

1. Como una *secuencia* de elementos de datos.
2. Como una *selección* de entre un conjunto de elementos de datos.
3. Como una *agrupación* repetitiva de elementos de datos. Cada entrada de elemento de datos que aparezca como parte de una secuencia, una selección o una repetición puede a su vez ser otro elemento de datos compuestos que necesite un mayor refinamiento en el diccionario.

Para ilustrar el uso del diccionario de datos, volvamos al DFD de nivel 2 y del proceso monitorizar el sistema de *HogarSeguro*, mostrado en la Figura 12.21. Refiriéndonos a la figura, especificamos el elemento de datos **número de teléfono.** ¿Qué es exactamente un número de teléfono? Puede ser un número local de siete dígitos, una extensión de **4** dígitos, o una secuencia de 25 dígitos para llamadas de larga distancia. El diccionario de datos nos proporciona una definición precisa de **número de teléfono** para el DFD en cuestión. Además, indica dónde y cómo se usa este elemento de datos y cualquier información adicional que le sea relevante. La entrada del diccionario de datos comienza de la siguiente forma:

nombre:	número de teléfono
alias:	ninguno
dónde se *usa/cómo* se *usa:*	*comprobar con ajustes iniciales* (salida) *marcar número* (entrada)

descripción:
número de teléfono = prefijo + número de acceso
prefijo = [* un número de cuatro dígitos que comience en 0 ó un número de cinco dígitos que comience por 0]
número de acceso = *secuencia numérica de cualquier tamaño*

Un número como el O1327 546381 queda descrito de esta forma.

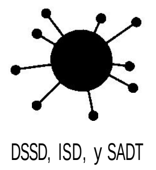
Para grandes sistemas basados en computadora el diccionario de datos crece rápidamente en tamaño y en complejidad. De hecho, es extremadamente difícil mantener manualmente el diccionario. Por esta razón, se deben usar herramientas CASE.

12.8 OTROS MÉTODOS CLÁSICOS DE ANÁLISIS

Durante los últimos años se han utilizado otros métodos valiosos de análisis de requisitos del software en la industria. Mientras que todos siguen los principios del análisis operacional tratados en el Capítulo 11, cada uno introduce una notación y heurística diferentes para construir el modelo de análisis. Una revisión de tres importantes métodos de análisis:

- *Desarrollo de sistemas estructurados de datos (DSED)*[WAR81, ORR81]
- *Desarrollo de sistemas Jackson (DSJ)*[JAC83]
- *Técnicas de análisis y diseño estructurado (TADE)* [ROS77, ROS85]

son presentadas dentro del sitio web SEPA para los lectores interesados en profundizar en el modelado del análisis.



DSSD, ISD, y SADT

RESUMEN

El análisis estructurado es el método más usado para el modelado de requisitos, utiliza el modelo de datos y el modelo de flujos para crear la base de un adecuado modelo de análisis. Utilizando el diagrama entidad-relación, el ingeniero del software crea una representación de todos los objetos de datos que son importantes para el sistema. Los sistemas de datos y flujo de control son la base de representación de la transformación de datos y control. **Al** mismo tiempo, estos métodos son usados para crear un modelo funcional del software y proveer-

se de un mecanismo para dividir funciones. Después, crea un modelo de comportamiento usando el diagrama de transición de estados y un modelo de contenido de los datos con un diccionario de datos. Las especificaciones de los procesos y del control proporcionan una elaboración adicional de los detalles.

La notación original para el análisis estructurado fue desarrollada para aplicaciones de procesamiento de datos convencionales, pero ahora hay ampliaciones que permiten aplicar el método a los sistemas de tiempo real.

El análisis estructurado está soportado por una larga lista de herramientas CASE que ayudan en la creación de cada elemento del modelo y también en el mantenimiento de la consistencia y de la corrección.

REFERENCIAS

[BRU88] Bruyn, W. et al., «ESML: An Extended Systems Modeling Language Based on the Data Flow Diagram», *ACM Software Engineering Notes*, vol. 13, n.º 1, Enero 1988, pp. 58-67.

[CHE77] Chen, P., *The Entity-Relationship Approach to Logical Database Design*, QED Information systems, 1977.

[DEM79] DeMarco, T., *Structured Analysis and System Specificaction*, Prentice-Hall, 1979.

[GAN82] Gane, T., y C. Sarson, *Structured Systems Analysis*, McDonnell Douglas, 1982.

[HAT87] Hatley, D.J., e I.A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, 1987.

[JAC83] Jackson, M.A., *System Development*, Prentice-Hall, 1983.

[ORR81] Orr, K.T., *Structured Requirements Definition*, Ken Orr & Associates, Inc., Topeka, KS, 1981.

[PAG80] Page-Jones, M., *The Practical Guide to Structured Systems Design*, Yourdon Press, 1980.

[ROS77] Ross, D., y K. Schoman, «Structured Analysis for Requirements Definitions», *IEEE Trans. Software Engineering*, vol. 3, n.º 1, Enero 1977, pp. 6-15.

[ROS84] Ross, D., «Applications and Extensions of SADT», *IEEE Computer*, vol. 18, n.º 4, Abril 1984, pp. 25-35.

[STE74] Stevens, W.P., G.J. Myers y L.L. Constantine, «Structured Design», *IBM Systems Journal*, vol. 13, n.º 2, 1974, pp. 115-139.

[TIL93] Tillman, G.A., *A Practical Guide to Logical Data Modeling*, McGraw-Hill, 1993.

[WAR81] Warnier, J.D., *Logical Construction of Programs*, Van Nostrand Reinhold, 1981.

[WAR85] Ward, P.T., y S.J. Mellor, *Structured Development for Real-Time Systems*, tres volúmenes, Yourdon Press, 1985.

[YOU78] Yourdon, E.N., y L.L. Constantine, *Structured Design*, Yourdon Press, 1978.

[YOU89] Yourdon, E.N., *Modern Structured Analysis*, Prentice Hall, 1990.

PROBLEMAS Y PUNTOS A CONSIDERAR

12.1. Obtenga al menos tres de las diferencias que se mencionan en la sección 12.1 y escriba un breve artículo que refleje el cambio a lo largo del tiempo de la concepción del análisis estructurado. En una sección de conclusiones, Sugiera formas en las que crea que cambiará el método en el futuro.

12.2. Se le pide que construya uno de los sistemas siguientes:

a. un sistema de registro en cursos basado en red para su universidad

b. un sistema procesamiento de transacciones basado en internet para un almacén de computadoras

c. un sistema simple de facturas para una empresa pequeña

d. un producto de software que sustituya a rolodex construido en un teléfono inalámbrico

e. un producto de libro de cocina automatizado construido en una cocina eléctrica o en un microondas

Seleccione el sistema que le sea de interés y desarrolle un diagrama entidad-relación que describa los objetos de datos, relaciones y atributos.

12.3. ¿Qué diferencia hay entre cardinalidad y modalidad?

12.4. Dibuje un modelo de nivel de contexto (DFD de nivel 0) para uno de los cinco sistemas que se listan en el problema 12.2. Escriba una descripción de procesamiento a nivel de contexto para el sistema.

12.5. Mediante el DFD de nivel de contexto desarrollado en el problema 12.4, desarrolle diagramas de flujo de datos de nivel 1 y nivel 2. Utilice un «analizador gramatical» en la descripción de procesamiento a nivel de contexto para que se inicie en ese tema. Recuerde especificar todo el flujo de información etiquetando todas las flechas entre burbujas. Utilice nombres significativos para cada transformación.

12.6. Desarrolle DFC, EC, EP y un diccionario de datos para el sistema que seleccionó en el problema 12.2. Intente construir su modelo tan completo como sea posible.

12.7. ¿Significa el concepto de continuidad del flujo de información que si una flecha de flujo aparece como entrada en el nivel 0, entonces en los subsiguientes niveles debe aparecer una flecha de flujo como entrada? Explique su respuesta.

12.8. Usando las ampliaciones de Ward y Mellor descritas en las figuras 12.13. ¿Cómo encaja la EC que se indica en la figura 12.13? Ward y Mellor no usan esa notación.

12.9. Usando las ampliaciones de Hatley y Pirbhai, rehaga el modelo de flujo contenido en la Figura 12.13. ¿Cómo encaja el modelo de control que se indica en la Figura 12.13? Hatley y Pirbhai no usan esa notación.

12.10. Describa con sus propias palabras un flujo de sucesos.

12.11. Desarrolle un modelo de flujo completo para el software de fotocopiadora discutido en la sección 12.5. Puede utilizar las ampliaciones de Ward y Mellor o las de Hatley y Pirbhai. Asegúrese de desarrollar para el sistema un diagrama de transición de estados detallado.

12.12. Complete la descripción de procesamiento para el software *HogarSeguro* que se ha presentado en la Figura 12.20; describiendo los mecanismo de interacción entre el usuario y el sistema. ¿Cambiará **su** información adicional los modelos de flujo de *HogarSeguro* que aparecen en este capítulo? Si es así, ¿cómo?

12.13. El departamento de obras públicas de un gran ciudad ha decidido desarrollar un *sistema de seguimiento y reparación de baches (SSRB)* basado en página web. Con los siguientes requisitos:

> Los ciudadanos pueden conectarse a la página e informar sobre la situación y la importancia del bache. A medida que se informa sobre cada bache, se le asigna un número de identificación y se guarda con la calle en la que se encuentra, **su** tamaño (en una escala de 1 a 10), su posición (en el medio, a un lado, etc.), **su** distrito (determinado a partir de la calle) y una prioridad de reparación (determinada a partir de su tamaño). A cada bache se le asocian datos de petición de obra, incluyendo la ubicación y el tamaño, la brigada, el equipamiento asignado, las horas de reparación, el estado del bache (obra en curso, reparado, reparación temporal, no reparado), la cantidad de material de relleno usado y el coste de la reparación (calculado con las horas dedicadas, el número de trabajadores, el material y el equipamiento usados). Finalmente, se crea un archivo de daños para mantener la información sobre los daños reportados debidos a la existencia del bache, incluyendo el nombre del ciudadano, **su** dirección, su número de teléfono, el tipo de daño y el coste de subsanamiento del daño. El sistema SSRB es un sistema interactivo.

Utilice el análisis estructurado para modelar SSRB.

12.14. Se va a desarrollar un sistema de procesamiento de textos basados en computadoras personales. Investigue durante algunas horas sobre el área de aplicación y lleve a cabo una reunión TFEA (Capítulo 11) con **sus** compañeros de clase para un modelo de requisitos del sistema utilizando el análisis estructurado (su profesor le ayudará a coordinarlo). Construya un modelo de requisitos del sistema mediante el análisis estructurado.

12.15. Se va a desarrollar el software para un videojuego. Proceda como en el problema 12.14.

12.16. Contacte con cuatro o cinco vendedores de herramientas **CASE** para análisis estructurado. Revise sus folletos y escriba un breve artículo que resuma las características generales y las que se distingan a unas herramientas de las otras.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Existen literalmente docenas de libros publicados sobre el análisis estructurado. Todos cubren el tema de forma adecuada, pero sólo unos pocos constituyen magníficos trabajos. El libro de DeMarco [DEM79] sigue siendo una buena introducción de la notación básica. Los libros de Hoffer et al. (*Modern Systems Analysis and Design*, Addison-Wesley, 2.ª ed., 1998), Kendall y Kendall (Systems Analysis and Design, 2.ª ed., Prentice Hall, 1998), Davis y Yen (*The Information System Consultant's Handbook: Systems Analysis and Design*, CRC Press, 1998), Modell (*A Professional's Cuide to Systems Analysis*, 2." ed., McGraw-Hill, 1996), Robertson and Robertson (*Complete Systems Analysis*, 2 vols., Dorset House, 1994), y Page-Jones (*The Practica1 Cuide to Structured Systems Design*, 2.ª ed., Prentice Hall, 1988) son referencias muy valiosas. El libro de Yourdon sobre este tema [YOU89] sigue siendo el tratado más extenso publicado hasta la fecha.

Para mayor énfasis en la ingeniería, [WAR85] y [HAT87] son los libros preferidos. En cualquier caso, Edwards (*Real-Time Structured Methods: Systems Analysis*, Wiley, 1993) trata el análisis de sistemas en tiempo real con gran profundidad, presentando diagramas de ejemplos Útiles sobre aplicaciones reales.

Se han desarrollado muchas variaciones sobre el análisis estructurado durante la última década. Cutts (*Structured Systems Analysis and Design Methodology*, Van Nostrand Reinhlod, 1990) y Hares (*SSADM for the Advanced Pructitioner*, Wiley, 1990) describe SSADM como una variación del análisis estructurado que se utiliza enormemente por toda Europa y Estados Unidos.

Flynn et al. (*Information Modelling: An International Perspective*, Prentice Hall, 1996), Reingruber y Gregory (*Data Modeling Handbook*, Wiley, 1995) y Tillman [TIL93] presentan manuales detallados para crear modelos de datos de calidad de industria. Kim y Salvatores («Comparing Data Modeling Formalisms», *Communications of the ACM*, June 1995) han escrito una comparación excelente de métodos de modelado de datos. Un libro interesante de Hay (*Data Modeling Patterns*, Dorset House, 1995) presenta los «patrones» comunes de modelos de datos que se encuentran en mucha empresas diferentes. En Kowal (*Behaviour Models: Specifying User's Expectations*, Prentice-Hall, 1992) se puede encontrar un tratamiento detallado del modelado de comportamiento.

Una amplía variedad de fuentes de información sobre el análisis estructurado están disponibles en internet. Una lista actualizada de páginas web que son interesantes sobre el concepto y métodos de análisis se encuentran en **http://www.pressman5.com**

CAPÍTULO

13 CONCEPTOS Y PRINCIPIOS DE DISEÑO

EL objetivo de los diseñadores es producir un modelo o representación de una entidad que se será construida a posteriori. Belady describe el proceso mediante el cual se desarrolla el modelo de diseño [BEL81]:

> En cualquier proceso de diseño existen dos fases importantes: la diversificación y la convergencia. La diversificación es la ***adquisición*** de un repertorio de alternativas, de un material primitivo de diseño: componentes, soluciones de componentes y conocimiento, todo dentro de catálogos, de libros de texto y en la mente. Durante la convergencia, el diseñador elige y combina los elementos adecuados y extraídos de este repertorio para satisfacer los objetivos del diseño, de la misma manera a como se establece en el documento de los requisitos, y de la manera en que se acordó con el cliente. La segunda fase es la ***eliminación*** gradual de cualquier configuración de componentes excepto de una en particular, y de aquí la creación del producto final.

La diversificación y la convergencia combinan intuición y juicio en función de la experiencia en construir entidades similares; un conjunto de principios y/o heurística que proporcionan la forma de guiar la evolución del modelo; un conjunto de criterios que posibilitan la calidad que se va **a** juzgar, y un proceso de iteración que por Último conduce a una representación final de diseño.

VISTAZO RÁPIDO

¿Qué es? El diseño es una representación significativa de ingeniería de algo que se va a construir. Se puede hacer el seguimiento basándose en los requisitos del cliente, y al mismo tiempo la calidad se puede evaluar y cotejar con el conjunto de criterios predefinidos para obtener un diseño «bueno». En el contexto de la ingeniería del software, el diseño se centra en cuatro áreas importantes de interés: datos, arquitectura, interfaces y componentes. En estas cuatro áreas se aplican los principios y conceptos que se abarcan en este capítulo

¿Quién lo hace? El ingeniero del software es quien diseña los sistemas basados en computadora, pero los conocimientos que se requieren en cada nivel de diseño funcionan de diferentes maneras. En el nivel de datos y de arquitectura, el diseño se centra en los patrones de la misma manera a como se aplican en la aplicación que se va a construir. En el nivel de la interfaz, es la ergonómica humana la que dicta nuestro enfoque de diseño. Y en el nivel de componentes, un «enfoque de programación» conduce a diseños de datos y procedimentales eficaces.

¿Por qué es importante? Si se construye una casa, ¿se hace sin un plano? Se correrían riesgos, se cometerían errores, habría un plano de casa sin sentido, con ventanas y puertas en sitios equivocados... un desastre. El software de computadora es considerablemente más complejo que una casa, de aquí que necesitemos un plano —el diseño—.

¿Cuáles son los pasos? El diseño comienza con el modelo de los requisitos. Se trabaja por transformar este modelo y obtener cuatro niveles de detalles de diseño: la estructura de datos, la arquitectura del sistema, la representación de la interfaz y los detalles a nivel de componentes. Durante cada una de las actividades del diseño, se aplican los conceptos y principios básicos que llevan a obtener una alta calidad.

¿Cuál es el producto obtenido? Por último se produce una especificación del diseño. La especificación se compone de los modelos del diseño que describen los datos, arquitectura, interfaces y componentes. Cada una de estas partes es lo que forma el producto obtenido del proceso de diseño.

¿Cómo puedo estar seguro de que lo he hecho correctamente? En cada etapa se revisan los productos del diseño del software en cuanto a claridad, corrección, finalización y consistencia, y se comparan con los requisitos y unos con otros.

El diseño del software, al igual que los enfoques de diseño de ingeniería en otras disciplinas, va cambiando continuamente a medida que se desarrollan métodos nuevos, análisis mejores y se amplia el conocimiento. Las metodologías de diseño del software carecen de la profundidad, flexibilidad y naturaleza cuantitativa que se asocian normalmente a las disciplinas de diseño de ingeniería más clásicas. Sin embargo, **sí** existen métodos para el diseño del software; también se dispone de calidad de diseño y se pueden aplicar notaciones de diseño. En este capítulo se explorarán los conceptos y principios fundamentales que se pueden aplicar a todo diseño de software. En los Capítulos **14,** 15, 16 y 22 se examinan diversos métodos de diseño de software en cuanto a la manera en que **se** aplican al diseño arquitectónico, de interfaz y a nivel de componentes.

13.1 DISEÑO DEL SOFTWARE E INGENIERÍA DEL SOFTWARE

El diseño del software se encuentra en el núcleo técnico de la ingeniería del software y se aplica independientemente del modelo de diseño de software que se utilice. Una vez que se analizan y especifican los requisitos del software, el diseño del software es la primera de las tres actividades técnicas – diseño, generación de código y pruebas — que se requieren para construir y verificar el software. Cada actividad transforma la información de manera que dé lugar por último a un software de computadora validado.

Cita:

Los milagros más comunes de la ingeniería del software son las transiciones desde el análisis hasta el diseño, y desde el diseño al código.

Richard Due

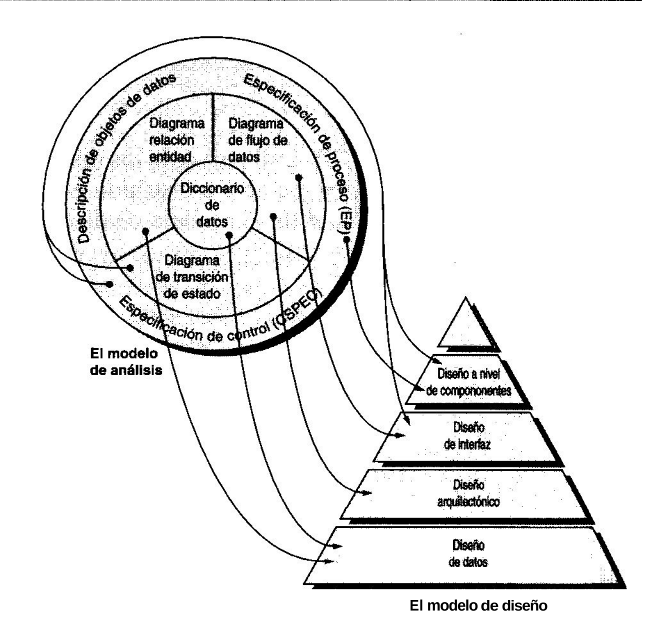


FIGURA 13.1. Conversión del modelo de análisis en un diseño de software.

Cada uno de los elementos del modelo de análisis (Capítulo 12) proporciona la información necesaria para crear los cuatro modelos de diseño que se requieren para una especificación completa de diseño. El flujo de información durante el diseño del software se muestra en la Figura **13.1.** Los requisitos del software, manifestados por los modelos de datos funcionales y de comportamiento, alimentan la tarea del diseño. Mediante uno de los muchos métodos de diseño (que se abarcarán en capítulos posteriores) la tarea de diseño produce un diseño de datos, un diseño arquitectónico, un diseño de interfaz y un diseño de componentes.

El *diseño de datos* transforma el modelo del dominio de información que se crea durante el análisis en las estructuras de datos que se necesitarán para implementar el software. Los objetos de datos y las relaciones definidas en el diagrama relación entidad y el contenido de datos detallado que se representa en el diccionario de datos proporcionan la base de la actividad del diseño de datos. Es posible que parte del diseño de datos tenga lugar junto con el diseño de la arquitectura del software. **A** medida que se van diseñando cada uno de los componentes del software, van apareciendo más detalles de diseño.

El *diseño arquitectónico* define la relación entre los elementos estructurales'principales del software, los patrones de diseño que se pueden utilizar para lograr los requisitos que se han definido para el sistema, y las restricciones que afectan a la manera en que se pueden aplicar los patrones de diseño arquitectónicos [SHA96]. La representación del diseño arquitectónico —el marco de trabajo de un sistema basado en computadora — puede derivarse de la especificación del sistema, del modelo de análisis y de la interacción del subsistema definido dentro del modelo de análisis.

El *diseño de la interfaz* describe la manera de comunicarse el software dentro de sí mismo, con sistemas que interoperan dentro de él y con las personas que lo utilizan. Una interfaz implica un flujo de información (por ejemplo, datos y/o control) y un tipo específico de comportamiento. Por tanto, los diagramas de flujo de control y de datos proporcionan gran parte de la información que se requiere para el diseño de la interfaz.

El *diseno a nivel de componentes* transforma los elementos estructurales de la arquitectura del software en una descripción procedimental de los componentes del software. La información que se obtiene de EP, EC y de DTE sirve como base para el diseño de los componentes.

La importancia del diseño del software se puede describir con una sola palabra —*calidad*—. El diseño es el lugar en donde se fomentará la calidad en la ingeniería del software. El diseño proporciona las representaciones del software que se pueden evaluar en cuanto a calidad. El diseño es la única forma de convertir exactamente los requisitos de un cliente en un producto o sistema de software finalizado. El diseño del software sirve como fundamento para todos los pasos siguientes del soporte del software y de la ingeniería del software. Sin un diseño, corremos el riesgo de construir un sistema inestable —un sistema que fallará cuando se lleven a cabo cambios; un sistema que puede resultar difícil de comprobar; y un sistema cuya calidad no puede evaluarse hasta muy avanzado el proceso, sin tiempo suficiente y con mucho dinero gastado en él—.

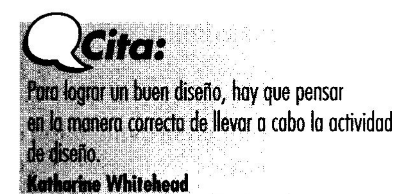
13.2 EL PROCESO DE DISEÑO

El diseño del software es un proceso iterativo mediante el cual los requisitos se traducen en un «plano» para construir el software. Inicialmente, el plano representa una visión holística del software. Esto es, el diseño se representa a un nivel alto de abstracción —un nivel que puede rastrearse directamente hasta conseguir el objetivo del sistema específico y según unos requisitos más detallados de comportamiento, funcionales y de datos—. A medida que ocurren las iteraciones del diseño, el refinamiento subsiguiente conduce a representaciones de diseño a niveles de abstracción mucho más bajos. Estos niveles se podrán rastrear aún según los requisitos, pero la conexión es más sutil.

13.2.1. Diseño y calidad del software

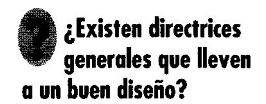
A lo largo de todo el proceso del diseño, la calidad de la evolución del diseño se evalúa con una serie de revisiones técnicas formales o con las revisiones de diseño abordadas en el Capítulo 8. McGlaughlin [MCG91] sugiere tres características que sirven como guía para la evaluación de un buen diseño:

- el diseño deberá implementar todos los requisitos explícitos del modelo de análisis, y deberán ajustarse a todos los requisitos implícitos que desea el cliente;
- el diseño deberá ser una guía legible y comprensible para aquellos que generan código y para aquellos que comprueban y consecuentemente, dan soporte al software;
- el diseño deberá proporcionar una imagen completa del software, enfrentándose a los dominios de comportamiento, funcionales y de datos desde una perspectiva de implementación.



Con el fin de evaluar la calidad de una representación de diseño, deberán establecerse los criterios técnicos para un buen diseño. Más adelante en este mismo Capítulo, se abordarán más detalladamente los criterios de calidad del diseño. Por ahora se presentarán las siguientes directrices:

1. Un diseño deberá presentar una estructura arquitectónica que (1) se haya creado mediante patrones de diseño reconocibles, (2) que esté formada por componentes que exhiban características de buen diseño (aquellas que se abordarán más adelante en este mismo capítulo), y (3) que se puedan implementar de manera evolutiva, facilitando así la implementación y la comprobación.
2. Un diseño deberá ser modular; esto es, el software deberá dividirse lógicamente en elementos que realicen funciones y subfunciones específicas.



3. Un diseño deberá contener distintas representaciones de datos, arquitectura, interfaces y componentes (módulos).
4. Un diseño deberá conducir a estructuras de datos adecuadas para los objetos que se van a implementar y que procedan de patrones de datos reconocibles.
5. Un diseño deberá conducir a componentes que presenten características funcionales independientes.
6. Un diseño deberá conducir a interfaces que reduzcan la complejidad de las conexiones entre los módulos y con el entorno externo.
7. Un diseño deberá derivarse mediante un método repetitivo y controlado por la información obtenida durante el análisis de los requisitos del software.

Estos criterios no se consiguen por casualidad. El proceso de diseño del software fomenta el buen diseño a través de la aplicación de principios de diseño fundamentales, de metodología sistemática y de una revisión cuidadosa.



Hay dos formas de construir un diseño de software: una forma es hacer un diseño tan simple que no existan obviamente deficiencias, y la otra es hacer un diseño tan complicado que no existan deficiencias obvias. La primera forma es mucho más difícil.
C.A.R. Hoare

13.2.2. La evolución del diseño del software

La evolución del diseño del software es un proceso continuo que ha abarcado las últimas cuatro décadas. El primer trabajo de diseño se concentraba en criterios para el desarrollo de programas modulares [DEN73] y métodos para refinar las estructuras del software de manera descendente [WIR71]. Los aspectos procedimentales de la definición de diseño evolucionaron en una filosofía denominada *programación estructurada* [DAH71, MIL72]. Un trabajo posterior propuso métodos para la conversión del flujo de datos [STE74] o estructura de datos [JAC75, WAR74] en una definición de diseño. Enfoques de diseño más recientes hacia la derivación de diseño proponen un método orientado a objetos. Hoy en día, se ha hecho hincapié en un diseño de software basado en la arquitectura del software [GAM95, BUS96, BRO98].

Independientemente del modelo de diseño que se utilice, un ingeniero del software deberá aplicar un conjunto de principios fundamentales y conceptos básicos para el diseño a nivel de componentes, de interfaz, arquitectónico y de datos. Estos principios y conceptos se estudian en la sección siguiente.

13.3 PRINCIPIOS DEL DISEÑO

El diseño de software es tanto un proceso como un modelo. El *proceso* de diseño es una secuencia de pasos que hacen posible que el diseñador describa todas los aspectos del software que se va construir. Sin embargo, es importante destacar que el proceso de diseño simplemente no es un recetario. Un conocimiento creativo, experiencia en el tema, un sentido de lo que hace que un software sea bueno, y un compromiso general con la calidad son factores críticos de éxito para un diseño competente.

El *modelo* de diseño es el equivalente a los planes de un arquitecto para una casa. Comienza representando la totalidad de todo lo que se va a construir (por ejemplo, una representación en tres dimensiones de la casa) y refina lentamente lo que va a proporcionar la guía para construir cada detalle (por ejemplo, el diseño de fontanería). De manera similar, el modelo de diseño que se crea para el software proporciona diversas visiones diferentes de software de computadora.

Los principios básicos de diseño hacen posible que el ingeniero del software navegue por el proceso de diseño. Davis [DAV95] sugiere un conjunto[1] de principios para el diseño del software, los cuales han sido adaptados y ampliados en la lista siguiente:

- *En el proceso de diseño no deberá utilizarse «orejeras».* Un buen diseñador deberá tener en cuenta enfoques alternativos, juzgando todos los que se basan en los requisitos del problema, los recursos disponibles para realizar el trabajo y los conceptos de diseño presentados en la Sección **13.4.**
- *El diseño deberá poderse rastrear hasta el modelo de análisis.* Dado que un solo elemento del modelo de diseño suele hacer un seguimiento de los múltiples requisitos, es necesario tener un medio de rastrear cómo se han satisfecho los requisitos por el modelo de diseño.
- *El diseño no deberá inventar nada que ya esté inventado.* Los sistemas se construyen utilizando un conjunto de patrones de diseño, muchos de los cuales probablemente ya se han encontrado antes. Estos patrones deberán elegirse siempre como una alternativa para reinventar. Hay poco tiempo y los recursos son limitados. El tiempo de diseño se deberá invertir en la representación verdadera de ideas nuevas y en la integración de esos patrones que ya existen.
- *El diseño deberá «minimizar la distancia intelectual» [DAV95] entre el software y el problema como si de la misma vida real se tratara.* Es decir, la estructura del diseño del software (siempre que sea posible) imita la estructura del dominio del problema.
- *El diseño deberá presentar uniformidad e integración.* Un diseño es uniforme si parece que fue una persona la que lo desarrolló por completo. Las reglas de estilo y de formato deberán definirse para un equipo de diseño antes de comenzar el trabajo sobre el diseño. Un diseño se integra si se tiene cuidado a la hora de definir interfaces entre los componentes del diseño.
- *El diseño deberá estructurarse para admitir cambios.* Los conceptos de diseño estudiados en la sección siguiente hacen posible un diseño que logra este principio.
- *El diseño deberá estructurarse para degradarse poco a poco, incluso cuando se enfrenta con datos, sucesos o condiciones de operación aberrantes.* Un software bien diseñado no deberá nunca explotar como una «bomba». Deberá diseñarse para adaptarse a circunstancias inusuales, y si debe terminar de funcionar, que lo haga de forma suave.



La consistencia del diseño y la uniformidad es crucial cuando se van a construir sistemas grandes. Se deberá establecer un conjunto de reglas de diseño para el equipo del software antes de comenzar a trabajar.

- *El diseño no es escribir código y escribir código no es diseñar.* Incluso cuando se crean diseños procedimentales para componentes de programas, el nivel de abstracción del modelo de diseño es mayor que el código fuente. Las únicas decisiones de diseño realizadas a nivel de codificación se enfrentan con pequeños datos de implementación que posibilitan codificar el diseño procedimental.
- *El diseño deberá evaluarse en función de la calidad mientras se va creando, no después de terminarlo.* Para ayudar al diseñador en la evaluación de la calidad se dispone de conceptos de diseño (Sección **13.4)** y de medidas de diseño (Capítulos 19 y 24).
- *El diseño deberá revisarse para minimizar los errores conceptuales (semánticos).* A veces existe la tendencia de centrarse en minucias cuando se revisa el diseño, olvidándose del bosque por culpa de los árboles. Un equipo de diseñadores deberá asegurarse de haber afrontado los elementos conceptuales principales antes de preocuparse por la sintaxis del modelo del diseño.

Referencia cruzada

En el Capítulo 8 se presentan las directrices para llevar a cabo revisiones de diseño efectivas.

[1] Aquí solo se destaca un pequeño subconjunto de los principios de diseño **de** Davis. Para mas información, véase [DAV95].

Cuando los principios de diseño descritos anteriormente se aplican adecuadamente, el ingeniero del software crea un diseño que muestra los factores de calidad tanto internos como externos [MEY88]. *Losfactores de calidad externos* son esas propiedades del software que pueden ser observadas fácilmente por los usuarios (por ejemplo, velocidad, fiabilidad, grado de corrección, usabilidad)[2]. Los*factores de calidad internos* tienen importancia para los ingenieros del software. Desde una perspectiva técnica conducen a un diseño de calidad alta. Para lograr los factores de calidad internos, el diseñador deberá comprender los conceptos de diseño básicos.

13.4 CONCEPTOS DEL DISEÑO

Durante las últimas cuatro décadas se ha experimentado la evolución de un conjunto de conceptos fundamentales de diseño de software. Aunque el grado de interés en cada concepto ha variado con los años, todos han experimentadoel paso del tiempo. Cada uno de ellos proporcionarán la base de donde el diseñador podrá aplicar los métodos de diseño más sofisticados. Cada uno ayudará al ingeniero del software a responder las preguntas siguientes:

- ¿Qué criterios se podrán utilizar para la partición del software en componentes individuales?
- ¿Cómo se puede separar la función y la estructura de datos de una representación conceptual del software?
- ¿Existen criterios uniformes que definen la calidad técnica de un diseño de software?

M.A. Jackson una vez dijo: «El comienzo de la sabiduría para un ingeniero del softwarees reconocerla diferencia entre hacer que un programa funcione y conseguir que lo haga correctamente».[JAC875] *Los* conceptos de diseño de software fundamentalesproporcionan el marco de trabajo necesario para conseguir q u e lo haga correctamente».

Cita:
La abstracción es una de las formas fundamentales en que los hombres se enfrentan con la complejidad.
Grady Booch

13.4.1. Abstracción

Cuando se tiene en consideración una solución modular a cualquier problema, se pueden exponer muchos *niveles de abstracción.* En el nivel más alto de abstracción, la solución se pone como una medida extensa empleando el lenguaje del entorno del problema. En niveles inferiores de abstracción, se toma una orientación más procedimental. La terminología orientada a problemas va emparejada con la terminología orientada a la implementación en un esfuerzo por solucionar el problema. Finalmente, en el nivel más bajo de abstracción, se establece la solución para poder implementarse directamente. Wasserman [WAS83] proporciona una definición útil:

> La noción psicológica de «abstracción» permite concentrarse en un problema a algún nivel de generalización **sin** tener en consideración los datos irrelevantes de bajo nivel; la utilización de la abstracción también permite trabajar con conceptos y términos que son familiares en el entorno del problema sin tener que transformarlos en una estructura no familiar.. .

Cada paso del proceso del software es un refinamiento en el nivel de abstracción de la solución del software. Durante la ingeniería del sistema, el software se asigna como un elemento de un sistema basado en computadora. Durante el análisis de los requisitos del software, la solución del software se establece en estos términos: «aquellos que son familiares en el entorno del problema». A medida que nos adentramos en el proceso de diseño, se reduce el nivel de abstracción. Finalmente el nivel de abstracción más bajo se alcanza cuando se genera el código fuente.

A medida que vamos entrando en diferentes niveles de abstracción, trabajamos para crear abstracciones procedimentales y de datos. Una *abstracción procedimental* es una secuencia nombrada de instrucciones que tiene una función específica y limitada. Un ejemplo de abstracción procedimental sería la palabra «abrir» para una puerta. «Abrir» implica una secuencia larga de pasos procedimentales (por ejemplo, llegar a la puerta; alcanzar **y** agarrar el pomo de la puerta; girar el pomo y tirar de la puerta; separarse al mover la puerta, etc.).



Como diseñador, trabaje mucho y duro poro derivar abstracciones tonto procedimentales como de datos que sirvan poro el problema que tengo en ese momento, pero que también se puedon volver a utilizar en otros situaciones.

Una *abstracción de datos* es una colección nombrada de datos que describe un objeto de datos (Capítulo 12). En el contexto de la abstracción procedimental *abrir,* podemos definir una abstracción de datos llamada **puerta.** Al igual que cualquier objeto de datos, la

[2] En el Capítulo 19 se presenta un estudio más detallado sobre los factores de calidad.

abstracción de datos para **puerta** acompañaría a un conjunto de atributos que describen esta puerta (por ejemplo, tipo de puerta, dirección de apertura, mecanismo de apertura, peso, dimensiones). Se puede seguir diciendo que la abstracción procedimental *abrir* hace uso de la información contenida en los atributos de la abstracción de datos **puerta.**

La *abstracción de control* es la tercera forma de abstracción que se utiliza en el diseño del software. Al igual que las abstracciones procedimentales y de datos, este tipo de abstracción implica un mecanismo de control de programa sin especificar los datos internos. Un ejemplo de abstracción de control es el semáforo de sincronización [KAI83] que se utiliza para coordinar las actividades en un sistema operativo. El concepto de abstracción de control se estudia brevemente en el Capítulo 14.

13.4.2. Refinamiento

El *refinamiento paso a paso* es una estrategia de diseño descendente propuesta originalmente por Niklaus Wirth [WIR71]. El desarrollo de un programa se realiza refinando sucesivamente los niveles de detalle procedimentales. Una jerarquía se desarrolla descomponiendo una sentencia macroscópica de función (una abstracción procedimental) paso a paso hasta alcanzar las sentencias del lenguaje de programación. Wirth [WIR71] proporciona una visión general de este concepto:

> En cada paso (del refinamiento), se descompone una o varias instrucciones del programa dado en instrucciones más detalladas. Esta descomposición sucesiva o refinamiento de especificaciones termina cuando todas las instrucciones se expresan en función de cualquier computadora subyacente o de cualquier lenguaje de programación.. . De **la** misma manera que se refinan las tareas, los datos también se tienen que refinar, descomponer o estructurar, y es natural refinar el programa y las especificaciones de los datos en paralelo.
>
> Todos **los** pasos del refinamiento implican decisiones de diseño. **Es** importante que.. . el programador conozca los criterios subyacentes (para decisiones de diseño) y **la** existencia de soluciones alternativas., .

El proceso de refinamiento de programas propuesto por Wirth es análogo al proceso de refinamiento y de partición que se utiliza durante el análisis de requisitos. La diferencia se encuentra en el nivel de detalle de implementación que se haya tomado en consideración, no en el enfoque.



Existe la tendencia de entrar en detalle inmediatamente, saltándose los pasos de refinamiento. Esto conduce o errores y omisiones y hace que el diseño seo más difícil de revisar. Realice el refinamiento paso a paso.

El refinamiento verdaderamente es un proceso de *elaboración.* Se comienza con una sentencia de función (o descripción de información) que se define a un nivel alto de abstracción. Esto es, la sentencia describe la función o información conceptualmente, pero no proporciona información sobre el funcionamiento interno de la información. El refinamiento hace que el diseñador trabaje sobre la sentencia original, proporcionando cada vez más detalles a medida que van teniendo lugar sucesivamente todos y cada uno de los refinamientos (elaboración).

13.4.3. Modularidad

El concepto de modularidad se ha ido exponiendo desde hace casi cinco décadas en el software de computadora. La arquitectura de computadora (descrita en la Sección 13.4.4) expresa la modularidad; es decir, el software se divide en componentes nombrados y abordados por separado, llamados frecuentemente *módulos,* que se integran para satisfacer los requisitos del problema.

Se ha afirmado que «la modularidad es el Único atributo del software que permite gestionar un programa intelectualmente» [MYE78]. El software monolítico (es decir, un programa grande formado por un Único módulo) no puede ser entendido fácilmente por el lector. La cantidad de rutas de control, la amplitud de referencias, la cantidad de variables y la complejidad global hará que el entendimiento esté muy cerca de ser imposible. Para ilustrar este punto, tomemos en consideración el siguiente argumento basado en observaciones humanas sobre la resolución de problemas.

Pensemos que $C(x)$ es una función que define la complejidad percibida de un problema x, y que $E(x)$ es una función que define el esfuerzo (oportuno) que se requiere para resolver un problema x. Para dos problemas p_1 y p_2, si

$$C(p_1) > C(p_2) \qquad (13.1a)$$

implica que

$$E(p_1) > E(p_2) \qquad (13.1b)$$

Cita:

Para el hombre siempre hay una solución fácil para cualquier problema —clara, plausible y equivocada—.

H.L Mencken

En general, este resultado es por intuición obvio. Se tarda más en resolver un problema difícil.

Mediante la experimentación humana en la resolución de problemas se ha averiguado otra característica interesante. Esta es,

$$C(p_1 + p_2) > C(p_1) + C(p_2) \qquad (13.2)$$

La ecuación (13.2) implica que la complejidad percibida de un problema que combina p_1 y p_2 es mayor

que la complejidad percibida cuando se considera cada problema por separado. Teniendo en cuenta la ecuación (13.2) y la condición implicada por la ecuación (13.1), se establece que

$$E(p_1 + p_2) > E(p_1) + E(p_2) \qquad (13.3)$$

Esto lleva a una conclusión: «divide y vencerás» —es más fácil resolver un problema complejo cuando se rompe en piezas manejables—. El resultado expresado en la ecuación (13.3) tiene implicaciones importantes en lo que respecta a la modularidad y al software. Es, de hecho, un argumento para la modularidad.



No modularice de más. La simplicidad de cada módulo se eclipsará con la complejidad de la integración.

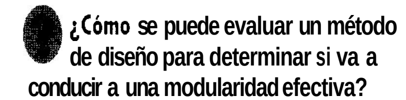
Es posible concluir de la ecuación (13.3) que si subdividimos el software indefinidamente, el esfuerzo que se requiere para desarrollarlo sería mínimo. Desgraciadamente, intervienen otras fuerzas, que hacen que esta conclusión sea (tristemente) falsa. Como muestra la Figura 13.2, el esfuerzo (coste) para desarrollar un módulo de software individual disminuye a medida que aumenta el número total de módulos. Dado el mismo conjunto de requisitos, tener más módulos conduce a un tamaño menor de módulo. Sin embargo, a medida que aumenta el número de módulos, también crece el esfuerzo (coste) asociado con la integración de módulos. Estas características conducen también a la curva total del coste o esfuerzo que se muestra en la figura. Existe un número M de módulos que daría como resultado un coste mínimo de desarrollo, aunque no tenemos la sofisticación necesaria para predecir M con seguridad.

Las curvas que se muestran en la Figura 13.2 proporcionan en efecto una guía útil cuando se tiene en consideración la modularidad. La modularidad deberá aplicarse, pero teniendo cuidado de estar próximo a M. Se deberá evitar modularizar de más o de menos. Pero, ¿cómo conocemos el entorno de M? ¿Cuánto se deberá modularizar el software? Para responder a estas preguntas se deberán comprender los conceptos de diseño que se estudiarán más adelante dentro de este capítulo.

Referencia cruzada

Los métodos de diseño se estudian en los Capítulos **14, 15, 16 y 22**.

Cuando se tiene en consideración la modularidad surge otra pregunta importante. ¿Cómo se define un módulo con un tamaño adecuado? La respuesta se encuentra en los métodos utilizados para definir los módulos dentro de un sistema. Meyer [MEY88] define cinco criterios que nos permitirán evaluar un método de diseño en relación con la habilidad de definir un sistema modular efectivo:



¿Cómo se puede evaluar un método de diseño para determinar si va a conducir a una modularidad efectiva?

Capacidad de descomposición modular. Si un método de diseño proporciona un mecanismo sistemático para descomponer el problema en subproblemas, reducirá la complejidad de todo el problema, consiguiendo de esta manera una solución modular efectiva.

Capacidad de empleo de componentes modulares. Si un método de diseño permite ensamblar los componentes de diseño (reusables) existentes en un sistema nuevo, producirá una solución modular que no inventa nada ya inventado.

Capacidad de comprensión modular. Si un módulo se puede comprender como una unidad autónoma (sin referencias a otros módulos) será más fácil de construir y de cambiar.

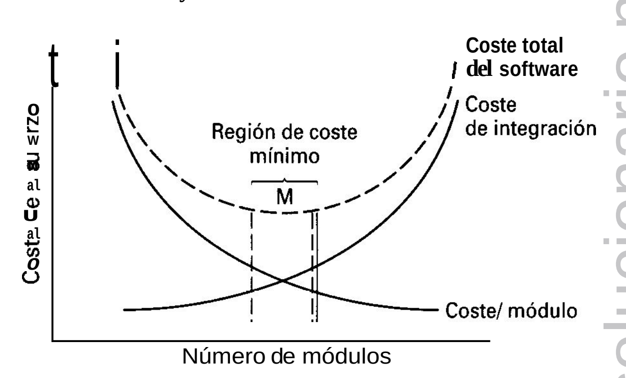


FIGURA 13.2. Modularidad y costes de software.

Continuidad modular. Si pequeños cambios en los requisitos del sistema provocan cambios en los módulos individuales, en vez de cambios generalizados en el sistema, se minimizará el impacto de los efectos secundarios de los cambios.

Protección modular. Si dentro de un módulo se produce una condición aberrante y sus efectos se limitan a ese módulo, se minimizará el impacto de los efectos secundarios inducidos por los errores.

Finalmente, es importante destacar que un sistema se puede diseñar modularmente, incluso aunque su implementación deba ser «monolítica». Existen situaciones (por ejemplo, software en tiempo real, software empotrado) en donde no es admisible que los subprogramas introduzcan sobrecargas de memoria y de velocidad por mínimos que sean (por ejemplo, subrutinas, procedimientos). En tales situaciones el software podrá y deberá diseñarse con modularidad como filosofía predominante. El código se puede desarrollar «en línea». Aunque el código fuente del programa puede no tener un aspecto modular a primera vista, se ha mantenido la filosofía y el programa proporcionará los beneficios de un sistema modular.

13.4.4. Arquitectura del software

La *arquitectura del software* alude a la «estructura global del software y a las formas en que la estructura proporciona la integridad conceptual de un sistema» [SHA95a]. En su forma más simple, la arquitectura es la estructura jerárquica de los componentes del programa (módulos), la manera en que los componentes interactúan y la estructura de datos que van a utilizar los componentes. Sin embargo, en un sentido más amplio, los «componentes» se pueden generalizar para representar los elementos principales del sistema y sus interacciones[3].



La STARS Software Architecture Technology Guide proporciona mbs información y recursos en la dirección **www-ast.tds-gn.lmco.com/arch/guide.html**

Un objetivo del diseño del software es derivar una representación arquitectónica de un sistema. Esta representación sirve como marco de trabajo desde donde se llevan a cabo actividades de diseño más detalladas. Un conjunto de patrones arquitectónicos permiten que el ingeniero del software reutilice los conceptos a nivel de diseño.

Cita:
Una arquitectura de software es el producto de trabajo de desarrollo que ofrece la mayor inclinación a invertir en lo que se refiere a calidad, planificación y coste.
Len Bass et al

Shaw **y** Garlan [SHA95a] describen un conjunto de propiedades que deberán especificarse como parte de un diseño arquitectónico:

Propiedades estructurales. Este aspecto de la representación del diseño arquitectónico define los componentes de un sistema (por ejemplo, módulos, objetos, filtros) y la manera en que esos componentes se empaquetan e interactúan unos con otros. Por ejemplo, **los** objetos se empaquetan para encapsular tanto los datos como el procesamiento que manipula **los** datos e interactúan mediante la invocación de métodos (Capítulo 20).

Propiedades extra-funcionales. La descripción del diseño arquitectónico deberá ocuparse de cómo la arquitectura de diseño consigue **los** requisitos para el rendimiento, capacidad, fiabilidad, seguridad, capacidad de adaptación y otras características del sistema.

Familias de sistemas relacionados. El diseño arquitectónico deberá dibujarse sobre patrones repetibles que se basen comúnmente en el diseño de familias de sistemas similares. En esencia, el diseño deberá tener la habilidad de volver a utilizar **los** bloques de construcción arquitectónicos.

Dada la especificación de estas propiedades, el diseño arquitectónico se puede representar mediante uno o más modelos diferentes [GAR95]. Los *modelos estructurales* representan la arquitectura como una colección organizada de componentes de programa. Los *modelos del marco de trabajo* aumentan el nivel de abstracción del diseño en un intento de identificar los marcos de trabajo (patrones) repetibles del diseño arquitectónico que se encuentran en tipos similares de aplicaciones. Los *modelos dinámicos* tratan los aspectos de comportamiento de la arquitectura del programa, indicando cómo puede cambiar la estructura o la configuración del sistema en función de los acontecimientos externos. **Los** *modelos de proceso* se centran en el diseño del proceso técnico de negocios que tiene que adaptar el sistema. Finalmente los *modelos funcionales* se pueden utilizar para representar la jerarquía funcional de un sistema.



Para representar el diseño arquitectónico se utilizan cinco tipos diferentes de modelos.

Se ha desarrollado un conjunto de *lenguajes de descripción arquitectónica* (LDAs) para representar los modelos destacados anteriormente [SHA95b]. Aunque se han propuesto muchos LDAs diferentes, la mayoría proporcionan mecanismos para describir los componentes del sistema y la manera en que se conectan unos con otros.

13.4.5. Jerarquía de control

La *jerarquía de control,* denominada también estructura de programa, representa la organización de los componentes de programa (módulos) e implica una jerarquía de control. No representa los aspectos procedimentales del software, ni se puede aplicar necesariamente a todos los estilos arquitectónicos.

Referencia cruzada

En el **Capítulo 14** se presenta un estudio detallado de estilos **y patrones** arquitectónicos.

[3] Por ejemplo, los componentes arquitectónicos de un sistema cliente/servidor se representan en un nivel de abstracción diferente. Para más detalles véase el Capítulo 28.

Para representar la jerarquía control de aquellos estilos arquitectónicos que se avienen a la representación se utiliza un conjunto de notaciones diferentes. El diagrama más común es el de forma de árbol (Fig. 13.3) que representa el control jerárquico para las arquitecturas de llamada y de retorno[4]. Sin embargo, otras notaciones, tales como los diagramas de Warnier-Orr [ORR77] y Jackson [JAC83] también se pueden utilizar con igual efectividad. Con objeto de facilitar estudios posteriores de estructura, definiremos una serie de medidas y términos simples. Según la Figura 13.3, la *profundidad* y la *anchura* proporcionan una indicación de la cantidad de niveles de control y el *ámbito de control* global, respectivamente. El *grado de salida* es una medida del número de módulos que se controlan directamente con otro módulo. El *grado de entrada* indica la cantidad de módulos que controlan directamente un módulo dado.

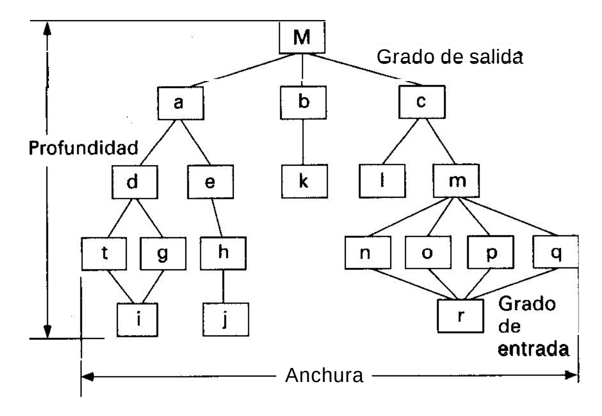


FIGURA 13.3. Terminologías de estructura para un estilo arquitectónico de llamada y retorno.

La relación de control entre los módulos se expresa de la manera siguiente: se dice que un módulo que controla otro módulo es *superior* a él, e inversamente, se dice que un módulo controlado por otro módulo es *subordinado* del controlador [YOU79]. Por ejemplo, en la Figura 13.3 el módulo ***M*** es superior a los módulos *a, b* y *c*. El módulo *h* está subordinado al módulo ***e*** y por Último está subordinado al módulo ***M***. Las relaciones en anchura (por ejemplo, entre los módulos *d* y ***e***), aunque en la práctica se puedan expresar, no tienen que definirse con terminología explícita.



Si desarrollamos un software orientado o objetos, no se aplicarán los medidos estructurales destacadas aquí. Sin embargo, se aplicarán otros (las que se abordan en lo Porte Cuarta).

La jerarquía de control también representa dos características sutiles diferentes de la arquitectura del software: visibilidad y conectividad. La *visibilidad* indica el conjunto de componentes de programa que un componente dado puede invocar o utilizar como datos, incluso cuando se lleva a cabo indirectamente. Por ejemplo, un módulo en un sistema orientado a objetos puede acceder al amplio abanico de objetos de datos que haya heredado, ahora bien solo utiliza una pequeña cantidad de estos objetos de datos. Todos los objetos son visibles para el módulo. La *conectividad* indica el conjunto de componentes que un componente dado invoca o utiliza directamente como datos. Por ejemplo, un módulo que hace directamente que otro módulo empiece la ejecución está conectado a él[5].

13.4.6. División estructural

Si el estilo arquitectónico de un sistema es jerárquico, la estructura del programa se puede dividir tanto horizontal como verticalmente. En la Figura 13.4.a la partición horizontal define ramas separadas de la jerarquía modular para cada función principal del programa. Lo *módulos de control,* representados con un sombreado más oscuro se utilizan para coordinar la comunicación entre ellos y la ejecución de las funciones. El enfoque más simple de la división horizontal define tres particiones - entrada, transformación de datos (frecuentemente llamado procesamiento) y salida —. La división horizontal de la arquitectura proporciona diferentes ventajas:

- proporciona software más fácil de probar
- conduce a un software más fácil de mantener
- propaga menos efectos secundarios
- proporciona software más fácil de ampliar



Como las funciones principales se desacoplan las unas de las otras, el cambio tiende a ser menos complejo y las extensiones del sistema (algo muy común) tienden a ser más fáciles de llevar a cabo sin efectos secundarios. En la parte negativa la división horizontal suele hacer que los datos pasen a través de interfaces de módulos y que puedan complicar el control global del flujo del programa (si se requiere un movimiento rápido de una función a otra).

[4] Una arquitectura de llamada y de retorno (Capítulo 14) es una estructura de programa clásica que descompone la función en una jerarquía de control en donde el programa «principal» invoca un número de componentes de programa que a su vez pueden invocar aún a otros componentes.

[5] En el Capítulo 20, exploraremos el concepto de herencia para el software orientado a objetos. Un componente de programa puede heredar una lógica de control y/o datos de otro componente sin referencia explícita en el código fuente. Los componentes de este tipo serán visibles, pero no estarán conectados directamente. Un diagrama de estructuras (Capítulo 14) indica la conectividad.

La división vertical (Fig. 13.4.b), frecuentemente llamada *factorización (factoring)* sugiere que dentro de la estructura de programa el control (toma de decisiones) y el trabajo se distribuyan de manera descendente. Los **módulos del** nivel superior deberán llevar a cabo funciones de control y no realizarán mucho trabajo de procesamiento. Los módulos que residen en la parte inferior de la estructura deberán ser los trabajadores, aquellos que realizan todas las tareas de entrada, proceso y salida.

La naturaleza del cambio en las estructuras de programas justifica la necesidad de la división vertical. En la Figura 13.4b se puede observar que un cambio en un módulo de control (parte superior de la estructura) tendrá una probabilidad mayor de propagar efectos secundarios a los módulos subordinados a él. Un cambio en el módulo de trabajador, dado su nivel bajo en la estructura, es menos probable que propague efectos secundanos. En general, los cambios en los programas de computadora giran alrededor del programa (es decir, su comportamiento básico es menos probable que cambie). Por esta razón las estructuras con división vertical son menos susceptibles a los efectos secundarios cuando se producen cambios y por tanto se podrán mantener mejor —un factor de calidad clave—.



los módulos de «trabajador» tienden a cambiar de forma más frecuente que los módulos de control. Si se colocan en la parte inferior de la estructura, se reducen los efectos secundarios (originados por el cambio).

13.4.7. Estructura de datos

La *estructura de datos* es una representación de la relación lógica entre elementos individuales de datos. Como la estructura de la información afectará invariablemente al diseño procedimental final, la estructura de datos es tan importante como la estructura de programa para la representación de la arquitectura del software.

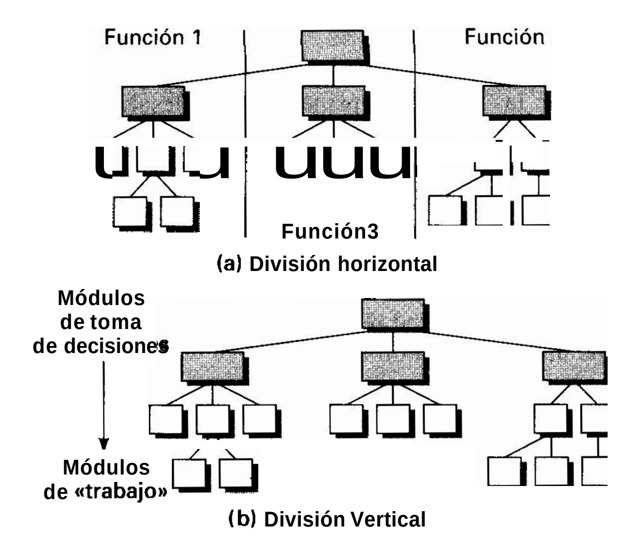
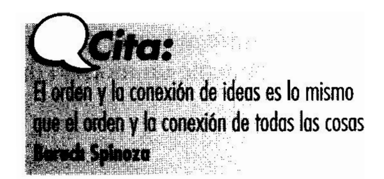


FIGURA 13.4. Partición estructural.

La estructura dicta las alternativas de organización, métodos de acceso, grado de capacidad de asociación y procesamiento de la información. Se han dedicado libros enteros (por ejemplo, [AHO83], [KRU84], [GAN89]) a estos temas, y un estudio más amplio sobre este tema queda fuera del ámbito de este libro. Sin embargo, es importante entender la disponibilidad de métodos clásicos para organizar la información y los conceptos que subyacen a las jerarquías de información.

La organización y complejidad de una estructura de datos están limitados Únicamente por la ingenuidad del diseñador. Sin embargo, existe un número limitado de estructuras de datos clásicas que componen los bloques de construcción para estructuras más sofisticadas.



Un *elemento escalar* es la estructura de datos más simple. Como su nombre indica, un elemento escalar representa un solo elemento de información que puede ser tratado por un identificador; es decir, se puede lograr acceso especificando un sola dirección en memoria. El tamaño y formato de un elemento escalar puede variar dentro de los límites que dicta el lenguaje de programación. Por ejemplo, un elemento escalar puede ser: una entidad lógica de un bit de tamaño; un entero o número de coma flotante con un tamaño de 8 a 64 bits; una cadena de caracteres de cientos o miles de bytes.

Cuando los elementos escalares se organizan como una lista o grupo contiguo, se forma un *vector secuencial*. Los vectores son las estructuras de datos más comunes y abren la puerta a la indexación variable de la información.

Cuando el vector secuencia1 se amplía a dos, tres y por último a un número arbitrario de dimensiones, se crea un *espacio n-dimensional*. El espacio n-dimensional más común es la matriz bidimensional. En muchos lenguajes de programación, un espacio n-dimensional se llama array.

Los elementos, vectores y espacios pueden estar organizados en diversos formatos. Una *lista enlazada* es una estructura de datos que organiza elementos escalares no contiguos, vectores o espacios de manera (llamados *nodos*) que les permita ser procesados como una lista. Cada nodo contiene la organización de datos adecuada (por ejemplo, un vector) o un puntero o más que indican la dirección de almacenamiento del siguiente nodo de la lista. Se pueden añadir nodos en cualquier punto de la lista para adaptar una entrada nueva en la lista.

Otras estructuras de datos incorporan o se construyen mediante las estructuras de datos fundamentales descritas anteriormente. Por ejemplo, una *estructura de datos jerárquica* se implementa mediante listas mul-

tienlazadas que contienen elementos escalares, vectores y posiblemente espacios n-dimensionales. Una estructurajerárquica se encuentra comúnmente en aplicaciones que requieren categorización y capacidad de asociación.



Invierta por lo menos todo el tiempo que necesite diseñando estructuras de datos, el mismo que pretende invertir diseñando algoritmos poro manipularlos. Si es así, se ahorrará mucha tiempo.

Es importante destacar que las estructuras de datos, al igual que las estructuras de programas, se pueden representar a diferentes niveles de abstracción. Por ejemplo, una pila es un modelo conceptual de una estructura de datos que se puede implementar como un vector o una lista enlazada. Dependiendo del nivel de detalle del diseño, los procesos internos de la **pila** pueden especificarse o no.

13.4.8. Procedimiento de software

La *estructura de programa* define la jerarquía de control sin tener en consideración la secuencia de proceso y de decisiones. El procedimiento de software se centra en el procesamiento de cada módulo individualmente. El procedimiento debe proporcionar una especificación precisa de procesamiento, incluyendo la secuencia de sucesos, los puntos de decisión exactos, las operaciones repetitivas e incluso la estructura/organización de datos.

Existe, por supuesto, una relación entre la estructura y el procedimiento. El procesamiento indicado para cada módulo debe incluir una referencia a todos los módulos subordinados al módulo que se está describiendo. Es decir, una representación procedimental del software se distribuye en capas como muestra la Figura 13.5[6].

13.4.9. Ocultación de información

El concepto de modularidad conduce a todos los diseñadores de software a formularse una pregunta importante: «¿Cómo se puede descomponer una solución de software para obtener el mejor conjunto de módulos?» El principio de *ocultación de información* [PAR72] sugiere que los módulos se caracterizan por las decisiones de diseño que (cada uno) oculta al otro. En otras palabras, los módulos deberán especificarse y diseñarse de manera que la información (procedimiento y datos) que está dentro de un módulo sea inaccesible a otros módulos que no necesiten esa información.

Ocultación significa que se puede conseguir una modularidad efectiva definiendo un conjunto de módulos independientes que se comunican entre sí intercambiando sólo la información necesaria para lograr la función del software. La abstracción ayuda a definir las entidades (o información).

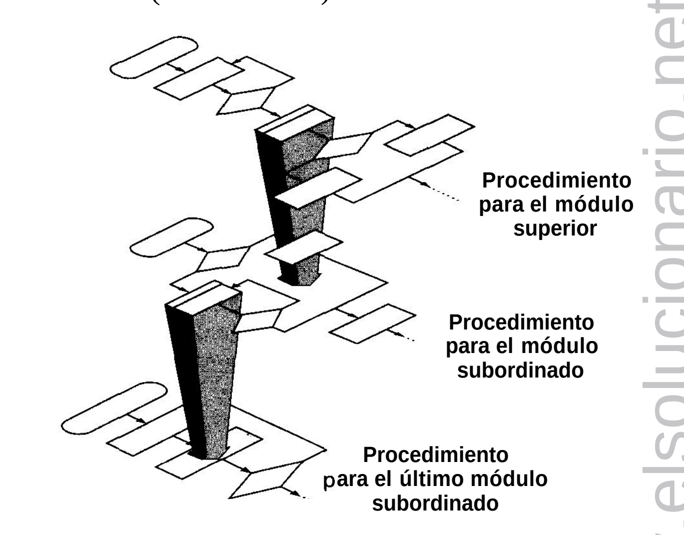


FIGURA 13.5. El procedimiento se distribuye en capas.

13.5. DISEÑO MODULAR EFECTIVO

Los conceptos fundamentales de diseño descritos en la sección anterior sirven para incentivar diseños modulares. De hecho, la modularidad se ha convertido en un enfoque aceptado en todas las disciplinas de ingeniería. Un diseño modular reduce la complejidad (véase la Sección 13.4.3), facilita los cambios (un aspecto crítico de la capacidad de mantenimiento del software), y da como resultado una implementación más fácil al fomentar el desarrollo paralelo de las diferentes partes de un sistema.

13.5.1. Independencia funcional

El concepto de *independenciafuncional* es la suma de la modularidad y de los conceptos de abstracción y ocultación de información. En referencias obligadas sobre el diseño del software, Pamas [PAR72] y Wirth [WIR71] aluden a las técnicas de refinamiento que mejoran la independencia de módulos. Trabajos posteriores de Stevens, Wyers y Constantine [STE74] consolidaron el concepto.



Un código es «determinante» si se describe con una sola oración —sujeto, verbo y predicada —.

La independencia funcional se consigue desarrollando módulos con una función «determinante» y una «aver-

[6] Esto no es verdad para todas las estructuras arquitectónicas, Por ejemplo, la estratificación jerárquica de procedimientos no se encuentra en arquitecturas orientadas a objetos.

sión» a una interacción excesiva con otros módulos. Dicho de otra manera, queremos diseñar el software de manera que cada módulo trate una subfunción de requisitos y tenga una interfaz sencilla cuando se observa desde otras partes de la estructura del programa. **Es** justo preguntarse por qué es importante la independencia. El software con una modularidad efectiva, es decir, módulos independientes, es más fácil de desarrollar porque la función se puede compartimentar y las interfaces se simplifican (tengamos en consideración las ramificaciones cuando el desarrollo se hace en equipo). Los módulos independientes son más fáciles de mantener (y probar) porque se limitan **los** efectos secundarios originados por modificaciones de diseño/código; porque se reduce la propagación de errores; y porque es posible utilizar módulos usables. En resumen, la independencia funcional es la clave para un buen diseño y el diseño es la clave para la calidad del software.

La independencia se mide mediante dos criterios cualitativos: la cohesión y el acoplamiento. La *cohesión* es una medida de la fuerza relativa funcional de un módulo. El *acoplamiento* es una medida de la independencia relativa entre los módulos.

13.5.2. Cohesión

La cohesión es una extensión natural del concepto de ocultación de información descrito en la Sección **13.4.8.** Un módulo cohesivo lleva a cabo una sola tarea dentro de un procedimiento de software, lo cual requiere poca interacción con los procedimientos que se llevan a cabo en otras partes de un programa. Dicho de manera sencilla, un módulo cohesivo deberá (idealmente) hacer una sola cosa.



La cohesión es una indicación cualitativa del grado que tiene un módulo para centrarse en una sola cosa.

La cohesión se puede representar como un «espectro». Siempre debemos buscar la cohesión más alta, aunque la parte media del espectro suele ser aceptable. La escala de cohesión no es lineal. Es decir, la parte baja de la cohesión es mucho «peor» que el rango medio, que es casi tan «bueno» como la parte alta de la escala. En la práctica, un diseñador no tiene que preocuparse de categorizar la cohesión en un módulo específico. Más bien, se deberá entender el concepto global, y así se deberán evitar los niveles bajos de cohesión al diseñar los códigos.

En la parte inferior (y no deseable) del espectro, encontraremos un módulo que lleva a cabo un conjunto de tareas que se relacionan con otras débilmente, si es que tienen algo que ver. Tales módulos se denominan *coincidentalmente cohesivos.* Un módulo que realiza tareas relacionadas lógicamente (por ejemplo, un módulo que produce todas las salidas independientemente del tipo) es *lógicamente cohesivo.* Cuando un módulo contiene tareas que están relacionadas entre **sí** por el hecho de que todas deben ser ejecutadas en el mismo intervalo de tiempo, el módulo muestra *cohesión temporal.*

Como ejemplo de baja cohesión, tomemos en consideración un módulo que lleva a cabo un procesamiento de errores de un paquete de análisis de ingeniería. El módulo es invocado cuando los datos calculados exceden los límites preestablecidos. Se realizan las tareas siguientes: **(1)** calcula los datos complementarios basados en los datos calculados originalmente; (2) produce un informe de errores (con contenido gráfico) en la estación de trabajo del usuario; **(3)** realiza los cálculos de seguimiento que haya pedido el usuario; **(4)** actualiza una base de datos, y (5) activa un menú de selección para el siguiente procesamiento. Aunque las tareas anteriores están poco relacionadas, cada una es una entidad funcional independiente que podrá realizarse mejor como un módulo separado. La combinación de funciones en un solo módulo puede servir sólo para incrementar la probabilidad de propagación de errores cuando se hace una modificación a alguna de las tareas procedimentales anteriormente mencionadas.

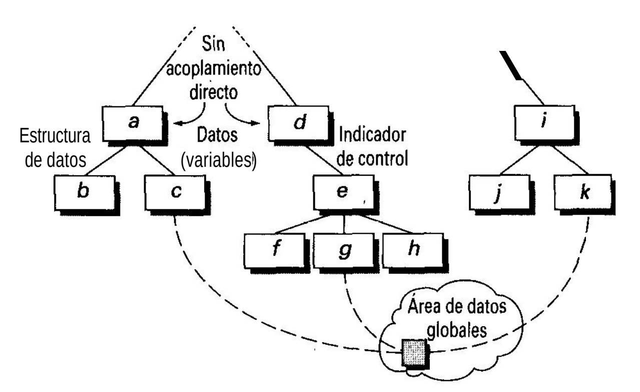


FIGURA 13.6. Tipos de acoplamiento.

Los niveles moderados de cohesión están relativamente cerca unos de otros en la escala de independencia modular. Cuando los elementos de procesamiento de un módulo están relacionados, y deben ejecutarse en un orden específico, existe *cohesión procedimental.* Cuando todos los elementos de procesamiento se centran en un área de una estructura de datos, tenemos presente una *cohesión de comunicación.* Una cohesión alta se caracteriza por un módulo que realiza una única tarea.



Si nos concentramos en una sola cosa durante el diseño a nivel de componentes, hay que realizarlo con cohesión.

Como ya se ha mencionado anteriormente, no es necesario determinar el nivel preciso de cohesión. **Más** bien, es importante intentar conseguir una cohesión alta y reconocer cuando hay poca cohesión para modificar el diseño del software y conseguir una mayor independencia funcional.

13.5.3. Acoplamiento

El acoplamiento es una medida de interconexión entre módulos dentro de una estructura de software. El acoplamiento depende de la complejidad de interconexión entre los módulos, el punto donde se realiza una entrada o referencia a un módulo, y los datos que pasan a través de la interfaz.

En el diseño del software, intentamos conseguir el acoplamiento más bajo posible. Una conectividad sencilla entre los módulos da como resultado un software **más** fácil de entender y menos propenso a tener un «efecto ola» [STE75] causado cuando ocurren errores en un lugar y se propagan por el sistema.

CLAVE

El acoplamiento es una indicación cualitativa del grado de conexión de un módulo con otros y con el mundo exterior.

La Figura 13.6 proporciona ejemplos de diferentes tipos de acoplamiento de módulos. Los módulos *a* y *d* son subordinados a módulos diferentes. Ninguno está relacionado y por tanto no ocurre un acoplamiento directo. El módulo *c* es subordinado al módulo *a* y se accede a él mediante una lista de argumentos por la que pasan los datos. Siempre que tengamos una lista convencional simple de argumentos (es decir, el paso de datos; la existencia de correspondencia uno a uno entre elementos), se presenta un acoplamiento bajo (llamado *acoplumiento de datos)* en esta parte de la estructura. Una variación del acoplamiento de datos, llamado *acoplamiento de marca (stamp),* se da cuando una parte de la estructura de datos (en vez de argumentos simples) se pasa a través de la interfaz. Esto ocurre entre los módulos *b* y *a*.



Sistemas altamente *acoplados* conducen *a* depurar verdaderas pesadillas. Evítelos.

En niveles moderados el acoplamiento **se** caracteriza por el paso de control entre módulos. El *acoplamiento de control* es muy común en la mayoría de los diseños de software y se muestra en la Figura 13.6 en donde un «indicador de control» (una variable que controla las decisiones en un módulo superior o subordinado) se pasa entre los módulos *d* **y** *e*.

Cuando los módulos están atados a un entorno externo al software se dan niveles relativamente altos de acoplamiento. Por ejemplo, la E/S ‘acoplaun módulo a dispositivos, formatos y protocolos de comunicación. El *acoplamiento externo* es esencial, pero deberá limitarse a unos pocos módulos en una estructura. También aparece un acoplamiento alto cuando varios módulos hacen referencia a un área global de datos. El *acoplamiento común,* tal y como se denomina este caso, se muestra en la Figura 13.8. Los módulos *c*, g y ***k*** acceden a elementos de datos en un área de datos global (por ejemplo, un archivo de disco o un área de memoria totalmente accesible). El módulo *c* inicializa el elemento. Más tarde el módulo g vuelve a calcular el elemento y lo actualiza. Supongamos que se produce un error y que g actualiza el elemento incorrectamente. Mucho más adelante en el procesamiento, el módulo ***k*** lee el elemento, intenta procesarlo y falla, haciendo que se interrumpa el sistema. El diagnóstico de problemas en estructuras con acoplamiento común es costoso en tiempo y es difícil. Sin embargo, esto no significa necesariamente que el uso de datos globales sea «malo». Significa que el diseñador del software deberá ser consciente de las consecuencias posibles del acoplamiento común y tener especial cuidado de prevenirse de ellos.

El grado más alto de acoplamiento, *acoplamiento de contenido,* se da cuando un módulo hace uso de datos o de información de control mantenidos dentro de los límites de otro módulo. En segundo lugar, el acoplamiento de contenido ocurre cuando se realizan bifurcaciones a mitad de módulo. Este modo de acoplamiento puede y deberá evitarse.

13.6 HEURÍSTICA DE DISEÑO PARA UNA MODULARIDAD EFECTIVA

Una vez que se ha desarrollado una estructura de programa, se puede conseguir una modularidad efectiva aplicando los conceptos de diseño que se introdujeron al principio de este capítulo. La estructura de programa se puede manipular de acuerdo con el siguiente conjunto de heurísticas:

I. Evaluar la «primera iteración» de la estructura de programa para reducir al acoplamiento y mejorar la cohesión. Una vez que se ha desarrollado la estructura del programa, se pueden explosionar o implosionar los módulos con vistas a mejorar la independencia del módulo. Un *módulo explosionado* se convierte en dos módulos o más en la estructura final de programa. *Un módulo implosionudo* es el resultado de combinar el proceso implicado en dos o más módulos.

Cita:

La idea de que las buenas técnicas (de diseño) restringen la creatividad es como decir que un artista puede pintar sin aprender nada sobre formas, o decir que un músico no necesita saber nada sobre teoría musical.

Marvin Zelkowitz et al

Un módulo explosionado se suele dar cuando existe un proceso común en dos o más módulos y puede redefinirse como un módulo de cohesión separado. Cuando se espera un acoplamiento alto, algunas veces se pueden implosionar los módulos para reducir el paso de control, hacer referencia a los datos globales y a la complejidad de la interfaz.

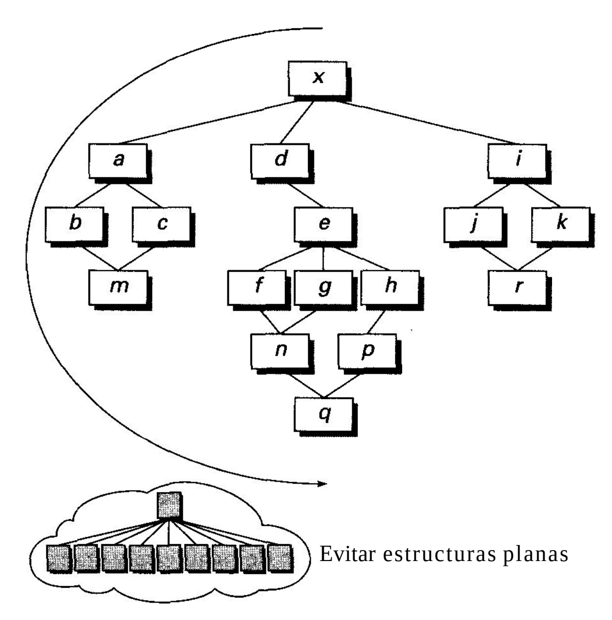


FIGURA 13.7. Estructuras de programa.

II. Intentar minimizar las estructuras con un alto grado de salida; esforzarse por la entrada a medida que aumenta la profundidad. La estructura que se muestra dentro de la nube en la Figura 13.7 no hace un uso eficaz de la factorización. Todos los módulos están «planos», al mismo nivel y por debajo de un solo módulo de control. En general, una distribución más razonable de control se muestra en la estructura de la derecha. La estructura toma una forma oval, indicando la cantidad de capas de control y módulos de alta utilidad a niveles inferiores.

III. Mantener el ámbito del efecto de un módulo dentro del ámbito de control de ese módulo. El *ámbito del efecto* de un módulo *e* se define como todos lo otros módulos que se ven afectados por la decisión tomada en el módulo ***e*.** El ámbito de control del módulo *e* se compone de todos los módulos subordinados y superiores al módulo *e*. En la Figura 13.7, si el módulo *e* toma una decisión que afecta al módulo *r*, tenemos una violación de la heurística III, porque el módulo *r* se encuentra fuera del ámbito de control del módulo *e*.

IV. *Evaluar las interfaces de los módulos para reducir la complejidad y la redundancia, y mejorar la consistencia.* La complejidad de la interfaz de un módulo es la primera causa de los errores del software. Las interfaces deberán diseñarse para pasar información de manera sencilla y deberán ser consecuentes con la función de un módulo. La inconsistencia de interfaces (es decir, datos aparentemente sin relacionar pasados a través de una lista de argumentos u otra técnica) es una indicación de poca cohesión. El módulo en cuestión deberá volverse a evaluar.

V. *Definir módulos cuya función se pueda predecir, pero evitar módulos que sean demasiado restrictivos.* Un módulo es predecible cuando se puede tratar como una caja negra; es decir, los mismos datos externos se producirán independientemente de los datos internos de procesamiento[7]. Los módulos que pueden tener «memoria» interna no podrán predecirse a menos que se tenga mucho cuidado en su empleo.

Un módulo que restringe el procesamiento a una sola subfunción exhibe una gran cohesión y es bien visto por el diseñador. Sin embargo, un módulo que restringe arbitrariamente el tamaño de una estructura de datos local, las opciones dentro del flujo de control o los modos de interfaz externa requerirá invariablemente mantenimiento para quitar esas restricciones.



En la dirección de internet **www.dacs.dtic.mil/techs/design/Design.ToC.html** se puede enconhor un informe detallado sobre los métodos de diseño de **software** entre los que se incluyen el estudio de todos los conceptos y principios de diseño que se abortan en este **capítulo**.

VI. Intentar conseguir módulos de «entrada controlada)), evitando «conexiones patológicas». Esta heurística de diseño advierte contra el acoplamiento de contenido. El software es más fácil de entender y por tanto más fácil de mantener cuando los módulos que ***se*** interaccionan están restringidos y controlados. Las *conexiones patológicas* hacen referencia a bifurcaciones o referencias en medio de un módulo.

[7] Un módulo de «caja negra. es una abstracción procedimental.

13.7 EL MODELO DEL DISEÑO

Los principios y conceptos de diseño abordados en este capítulo establecen las bases para la creación del modelo de diseño que comprende representaciones de datos, arquitectura, interfaces y componentes. Al igual que en el modelo de análisis anterior al modelo, cada una de estas representaciones de diseño se encuentran unidas unas a otras y podrán sufrir un seguimiento hasta los requisitos del software.

En la Figura 13.1, el modelo de diseño se representó como una pirámide. El simbolismo de esta forma es importante. Una pirámide es un objeto extremadamente estable con una base amplia y con un centro de gravedad bajo. Al igual que la pirámide, nosotros queremos crear un diseño de software que sea estable. Crearemos un modelo de diseño que se tambalee fácilmente con vientos de cambio al establecer una base amplia en el diseño de datos, mediante una región media estable en el diseño arquitectónico y de interfaz, y una parte superior aplicando el diseño a nivel de componentes.

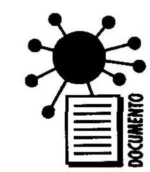
Los métodos que conducen a la creación del modelo de diseño se presentan en los Capítulos **14,** 15, 16 y 22 (para sistemas orientados a objetos). Cada método permite que el diseñador cree un diseño estable que se ajuste a los conceptos fundamentales que conducen a un software de alta calidad.

13.8 DOCUMENTACIÓN DEL DISEÑO

La *Especificación del diseño* aborda diferentes aspectos del modelo de diseño y se completa a medida que el diseñador refina su propia representación del software. En primer lugar, ***se*** describe el ámbito global del esfuerzo realizado en el diseño. La mayor parte de la información que se presenta aquí se deriva de la *Especificación del sistema* y del modelo de análisis *(Especificación de los requisitos del software).*

A continuación, se especifica el diseño de datos. Se definen también las estructuras de las bases de datos, cualquier estructura externa de archivos, estructuras internas de datos y una referencia cruzada que conecta objetos de datos con archivos específicos.

El diseño arquitectónico indica cómo se ha derivado la arquitectura del programa del modelo de análisis. Además, para representar la jerarquía del módulo se utilizan gráficos de estructuras.



Especificación del diseño del software

Se representa el diseño de interfaces internas y externas de programas y se describe un diseño detallado de la interfaz hombre-máquina. En algunos casos, se podrá representar un prototipo detallado del IGU.

Los componentes —elementos de software que se pueden tratar por separado tales como subrutinas, funciones o procedimientos— se describen inicialmente con una narrativa de procesamienfo en cualquier idioma (Castellano, Inglés). La narrativa de procesamiento explica la función procedimental de un componente (módulo). Posteriormente, se utiliza una herramienta de diseño procedimental para convertir esa estructura en una descripción estructural.

La *Especificación del diseño* contiene una referencia cruzada de requisitos. El propósito de esta referencia cruzada (normalmente representada como una matriz simple) es: (1) establecer que todos los requisitos se satisfagan mediante el diseño del software, y (2) indicar cuales son los componentes críticos para la implementación de requisitos específicos.

El primer paso en el desarrollo de la documentación de pruebas también se encuentra dentro del documento del diseño. Una vez que se han establecido las interfaces y la estructura de programa podremos desarrollar las líneas generales para comprobar los módulos individuales y la integración de todo el paquete. En algunos casos, esta sección se podrá borrar de la *Especificación del diseño.*

Las restricciones de diseño, tales como limitaciones físicas de memoria o la necesidad de una interfaz externa especializada, podrán dictar requisitos especiales para ensamblar o empaquetar el software. Consideraciones especiales originadas por la necesidad de superposición de programas, gestión de memoria virtual, procesamiento de alta velocidad u otros factores podrán originar modificaciones en diseño derivadas del flujo o estructura de la información. Además, esta sección describe el enfoque que se utilizará para transferir software al cliente.

La última sección de la *Especificación del diseño* contiene datos complementarios. También se presentan descripciones de algoritmos, procedimientos alternativos, datos tabulares, extractos de otros documentos y otro tipo de información relevante, todos mediante notas especiales o apéndices separados. Será aconsejable desarrollar un *Manual preliminar de Operaciones/Instalación* e incluirlo como apéndice para la documentación del diseño.

RESUMEN

El diseño es el núcleo técnico de la ingeniería del software. Durante el diseño se desarrollan, revisan y documentan los refinamientos progresivos de la estructura de datos, arquitectura, interfaces y datos procedimentales de los componentes del software. El diseño da como resultado representaciones del software para evaluar la calidad.

Durante las cuatro últimas décadas se han propuesto diferentes principios y conceptos fundamentales del diseño del software. Los principios del diseño sirven de guía al ingeniero del software a medida que avanza el proceso de diseño. Los conceptos de diseño proporcionan los criterios básicos para la calidad del diseño.

La modularidad (tanto en el programa como en los datos) y el concepto de abstracción permiten que el diseñador simplifique y reutilice los componentes del software. El refinamiento proporciona un mecanismo para representar sucesivas capas de datos funcionales. El programa y la estructura de datos contribuyen a tener una visión global de la arquitectura del software, mientras que el procedimiento proporciona el detalle necesario para la implementación de los algoritmos. La ocultación de información y la independencia funcional **pro**porcionan la heurística para conseguir una modularidad efectiva.

Concluiremos nuestro estudio de los fundamentos del diseño con las palabras de Glendord Myers [MYE78]:

> Intentamos resolver el problema dándonos prisa en el proceso de diseño de forma que quede el tiempo suficiente **hasta** el final del proyecto como para descubrir los errores que se cometieron por correr en el proceso de diseño...

La moraleja es: ¡No te precipites durante el diseño! Merece la pena esforzarse por un buen diseño.

No hemos terminado nuestro estudio sobre el diseño. En los capítulos siguientes, se abordan los métodos de diseño. Estos métodos combinados con los fundamentos de este capítulo, forman la base de una visión completa del diseño del software.

REFERENCIAS

[AHO83] Aho, A.V.J., Hopcroft, y J. Ullmann, *Data Structures and Algorithms,* Addison-Wesley, 1983.

[BAS89] Bass, L., P. Clements y R. Kazman, *Software Architecture in Practice,* Addison-Wesley, 1998.

[BEL81] Belady, L., Foreword to *Software Design: Methods and Techniques* (L.J. Peters, autor), Yourdon Press, 1981.

[BRO98] Brown, W.J. et al., *Anti-Patterns,* Wiley, 1998.

[BUS96] Buschmann, F. et al., *Pattern-Oriented Software Architecture,* Wiley, 1996.

[DAV95] Davis, A., *201 Principles of Software Development,* McGraw-Hill, 1995.

[DEN73] Dennis, J., «Modularity», in *Advanced Course on Software Engineering,* F.L. Bauer, ed., Springer-Verlag, Nueva York, 1973, pp. 128-182.

[GAM95] Gamma, E., et al, *Design Patterns,* Addison-Wesley, 1995.

[GAN89] Gannet, G., *Handbook of Algorithms and Data Structures,* 2.ª ed, Addison-Wesley, 1989.

[GAR95] Garlan, D., y M. Shaw, «An Introduction to Software Architecture», *Advances in Software Engineering and Knowledge Engineering,* vol. 1, V. Ambriola y G. Tortora (eds.), World Scientific Publishing Company, 1995.

[JAC75] Jackson, M.A., *Principles of Program Design,* Academic Press, 1975.

[JAC83] Jackson, M.A., *System Development,* Prentice-Hall, 1983.

[KAI83] Kaiser, S.H., *The Design of Operating Systems for Small Computer Systems,* Wiley-Inerscience, 1983, pp. 594 ss.

[JAC92] Jacobson, I., *Object-Oriented Software Engineering,* Addison-Wesley, 1992.

[KRU84] Kruse R.L., *Data Structures and Program Design,* Prentice-Hall, 1984.

[MCG91] McGlaughlin, R., «Some Notes on Program Design», *Software Engineering Notes,* vol. 16, n.º 4, Octubre 1991, pp. 53-54.

[MYE78] Myers, G., *Composite Structured Design,* Van Nostrand, 1978.

[MEY88] Meyer, B., *Object-Oriented Software Construction,* Prentice-Hall, 1988.

[MIL72] Mills, H.D., «Mathematical Foundations for Structured Programming», Technical Report FSC71-6012, IBM Corp., Federal Systems Division, Gaithersburg, Maryland, 1972.

[NAS73] Nassi, I., y B. Shneiderman, «Flowchart Techniques for Structured Programming», *SIGPLAN Notices,* ACM, Agosto 1973.

[ORR77] Orr, K.T., *Structured Systems Development,* Yourdon Press, Nueva York, 1977.

[PAR72] Parnas, D.L., «On Criteria to be used in Decomposing Systems into Modules», *CACM,* vol. 14, n.º 1, Abril 1972, pp. 221-227.

[ROS75] Ross, D., J. Goodenough y C. Irvine, «Software Engineering: Process, Principies and Goals», *IEEE Computer,* vol. 8, n.º 5, Mayo 1975.

[SHA95a] Shaw, M., y D. Garlan, «Formulations and Formalisms in Software Architecture», *Volume 1000-Lecture Notes in Computer Science,* Springer Verlag, 1995.

[SHA95b] Shaw, M. et al., «Abstractions for Software Architecture and Tools to Suppori Them», *IEEE Truns. Software Engineering,* vol. 21, n.° 4, Abril 1995, pp. 314-335.

[SHA96] Shaw, M., y D. Garlan, *Software Architecture,* Prentice-Hall, 1996.

[SOM96] Sommerville, I., *Software Engineering,* 3.ª ed., Addison-Wesley, 1989.

[STE74] Stevens, W., G. Myers y L. Constantine, «Structured Design», *IBM Systems Journal,* vol. 13, n.° 2, 1974, pp. 115-139.

[WAR74] Warnier, J., *Logical Constructionof Programs,* Van Nostrand-Reinhold, 1974.

[WAS83] Wasserman, A., «Information System Design Methodology», in *Software Design Techniques* (P. Freeman y A. Wasserman, eds.), 4.ª ed., IEEE Computer Society Press, 1983, p. 43.

[WIR71] Wirth, N., «Program Development by Stepwise Refinement», *CACM,* vol. 14, n.° 4, 1971, pp. 221-227.

[YOU79] Yourdon, E., y L. Constantine, *Structured Design,* Prentice-Hall, 1979.

PROBLEMAS Y PUNTOS A CONSIDERAR

13.1. Cuando se «escribe» un programa, ¿se está diseñando software? ¿En qué se diferencia el diseño del software de la codificación?

13.2. Desarrolle tres principios de diseño adicionales que añadir a los destacados en la Sección 13.3.

13.3. Proporcione ejemplos de tres abstracciones de datos y las abstracciones procedimentales que se pueden utilizar para manipularlos.

13.4. Aplique un «enfoque de refinamiento paso a paso» para desarrollar tres niveles diferentes de abstracción procedimental para uno o más de los programas que se muestran a continuación:

a. Desarrolle un dispositivo para rellenar los cheques que, dada la cantidad numérica en pesetas, imprima la cantidad en letra tal y como se requiere normalmente.

b. Resuelva iterativamente las raíces de una ecuación transcendental.

c. Desarrolle un simple algoritmo de planificación *round-robin* para un sistema operativo

13.5. ¿Es posible que haya algún caso en que la ecuación **(13.2)** no sea verdad? ¿Cómo podría afectar este caso a la modularidad?

13.6. ¿Cuándo deberá implementarse un diseño modular como un software monolítico? ¿Cómo se puede llevar a cabo? ¿Es el rendimiento la única justificación para la implementación de software monolítico?

13.7. Desarrolle al menos cinco niveles de abstracción para uno de los problemas de software siguientes:

a. un vídeo-juego de su elección.

b. un software de transformación en tres dimensiones para aplicaciones gráficas de computador.

c. un intérprete de lenguajes de programación.

d. un controlador de un robot con dos grados de libertad.

e. cualquier problema que hayan acordado usted y su profesor.

Conforme disminuye el grado de abstracción, su foco de atención puede disminuir de manera **que** en la última abstracción (código fuente) solo se necesite describir una única tarea.

13.8. Obtenga el trabajo original de Parnas [PAR72] y resuma el ejemplo de software que utiliza el autor para ilustrar la descomposición en módulos de un sistema. ¿Cómo se utiliza la ocultación de información para conseguir la descomposición?

13.9. Estudie la relación entre el concepto de ocultación de información como atributo de modularidad eficaz y el concepto de independencia del módulo.

13.10. Revise algunos de los esfuerzos que haya realizado recientemente en desarrollar un software y realice un análisis de **los** grados de cada módulo (con una escala ascendente del 1 al 7). Obtenga los mejores y los peores ejemplos.

13.11. Un grupo de lenguajes de programación de alto nivel soporta el procedimiento interno como construcción modular. ¿Cómo afecta esta construcción al acoplamiento y a la ocultación de información?

13.12. ¿Qué relación tienen los conceptos de acoplamiento y de movilidad del software? Proporcione ejemplos que apoyen esta relación.

13.13. Haga un estudio de la manera en que ayuda la partición estructural para ayudar a mantener el software.

13.14. ¿Cuál es el propósito de desarrollar una estructura de programa descompuesta en factores?

13.15. Describa el concepto de ocultación de información con sus propias palabras.

13.16. ¿Por qué es una buena idea mantener el efecto de alcance de un módulo dentro de su alcance de control?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Donald Norman ha escrito dos libros *(The Design of Everyday Things,* Doubleday, 1990, y *The Psychology of Everyday Things,* Harpercollins, 1988) que se han convertido en clásicos de literatura de diseño y es una lectura «obligada» para todos los que diseñan cualquier cosa que el ser humano va utilizar. Adams *(Conceptual Blockbusting,* 3." ed. Addison-Wesley, 1986) ha escrito un libro que es una lectura esencial para los diseñadores que quieren ampliar **su** forma de pensar. Finalmente un texto clásico de Polya *(How to Solve It,* 2." ed., Princeton University Press, 1988) proporciona un proceso genérico de resolver problemas que puede servir de ayuda a los diseñadores cuando se enfrentan con problemas complejos.

Siguiendo la misma tradición, Winograd et al. *(Bringing to Software,* Addison-Wesley, 1996) aborda los diseños del software que funcionan, los que no funcionan y las razones. Un libro fascinante editado por Wixon y Ramsey *(Field Methods Casebookfor Software Design,* Wiley, 1996) sugiere los «métodos de investigación de campos» (muy similares a los que utilizan los antropólogos) para entender cómo realizan los usuarios el trabajo que realizan y entonces diseñar el software que cubra sus necesidades. Beyer y Holtzblatt *(Contextual Design: A Customer-Centrered Approach to Systems,* Academic Press, 1997) ofrecen otra visión del diseño de software que integra al cliente/usuario en todos los aspectos del proceso de diseño del software.

McConnell *(Code Complete,* Microsoft Press, 1993) presenta **un** estudio excelente de los aspectos prácticos para el diseño de software de computadora de alta calidad. Robertson *(Simple Program Design,* 3.ª ed., Boyd & Fraser Publishing) presenta un estudio introductorio de diseño del software que es útil para aquellos que empiezan a estudiar sobre el tema.

Dentro de una antología editada por Freeman y Wasserman *(Software Design Techniques,* 4.ª ed., IEEE, 1983) se encuentra un estudio histórico excelente de trabajos importantes sobre diseño del software. Este trabajo de enseñanza reimprime muchos de los trabajos clásicos que han formado la base de las tendencias actuales en el diseño del software. Buenos estudios sobre los fundamentos del diseño de software se pueden encontrar en los libros de Myers [MYE78], Peters *(Software Design: Methods and Techniques,* Yourdon Press, Nueva York, 1981), Macro *(Software Engineering: Concepts and Management,* Prentice-Hall, 1990) y Sommerville *(Software Engineering,* Addison-Wesley, 5.ª ed., 1995).

Tratamientos matemáticamente rigurosos del software de computadora se pueden encontrar en los libros de Jones *(Software Development: A Rigourous Approach,* Prentice-Hall, 1980), Wulf *(Fundamental Structures of Computer Science,* Addison-Wesley, 1981) y Brassard y Bratley *(Fundamental of Algorithmics,* Prentice-Hall, 1995).

Todos estos libros ayudan a proporcionar el fundamento teórico necesario para comprender el software de computadora.

Kruse *(Data Structures and Program Design,* Prentice-Hall, 1994) y Tucker et al. *(Fundamental of Computing II: Abstraction, Data Structures, and Large Software Systems,* McGraw-Hill, 1995) presentan una información valiosa sobre estructuras de datos. Las medidas de calidad de diseño presentadas desde una perspectiva técnica y de gestión son abordadas por Card y Glass *(Measuring Design Quality,* Prentice-Hall, 1990).

Una amplia variedad de fuentes de información sobre el diseño del software y otros temas relacionados están disponibles en Intemet. Una lista actualizada de referencias relevantes para los conceptos y métodos de diseño se puede encontrar en **http://www.pressman5.com**

CAPÍTULO

14 DISEÑO ARQUITECTÓNICO

EL diseño se ha descrito como un proceso multifase en el que se sintetizan representaciones de la estructura de los datos, la estructura del programa, las características de la interfaz y los detalles procedimentales desde los requisitos de la información. Esta descripción es ampliada por Freeman [FRE80]:

> El diseño es una actividad en la que se toman decisiones importantes, frecuentemente de naturaleza estructural. Comparte con la programación un interés por la abstracción de la representación de la información y de las secuencias de procesamiento, pero el nivel de detalle es muy diferente en ambos casos. El diseño construye representaciones coherentes y bien planificadas de los programas, concentrándose en las interrelaciones de los componentes de mayor nivel y en las operaciones lógicas implicadas en **los** niveles inferiores...

Como dijimos en el capítulo anterior, el diseño está dirigido por la información. Los métodos de diseño del software se obtienen del estudio de cada uno de los tres dominios del modelo de análisis. El dominio de los datos, el funcional y el de comportamiento sirven de directriz para la creación del diseño.

En este capítulo se introducen los métodos requeridos para la creación de «representaciones coherentes y bien planeadas» de los datos y las capas arquitectónicas del modelo de diseño. El objetivo es proporcionar un enfoque sistemático para la obtención del diseño arquitectónico —el anteproyecto preliminar desde el que se construye el software—.

VISTAZO RÁPIDO

¿Qué es? El diseño arquitectónico representa la estructura de los datos y los componentes del programa que se requieren para construir un sistema basado en computadora. Constituye el estilo arquitectónico que tendrá el sistema, la estructura y las propiedades de los componentes que ese sistema comprende, y las interrelaciones que tienen lugar entre todos los componentes arquitectónicos del sistema.

¿Quién lo hace? Aunque los ingenieros del software pueden diseñar tanto los datos como la arquitectura, cuando se trata de construir sistemas grandes y complejos, el trabajo es, a menudo, asignado a especialistas. El diseñador de una base de datos o un almacén de datos crea la arquitectura de datos para el sistema. El «arquitecto del sistema» selecciona un estilo arquitectónico apropiado a los requisitos derivados durante el análisis de la ingeniería del sistema y de los requisitos del software.

¿Por qué es importante? En la sección *Vistazo rápido* del capítulo anterior preguntamos: «No serías capaz de intentar construir una casa sin un plano, ¿verdad?» Tú tampoco comenzarías el esbozo del plano por los bosquejos del diseño de tuberías de la casa. Necesitarías ver la foto completa —la casa en sí misma— antes de preocuparte por los detalles. Esto es lo que el diseño arquitectónico hace —proporciona la foto completa y asegura que lo estás haciendo bien—.

¿Cuáles son los pasos? El diseño arquitectónico comienza con el diseño de datos y después procede a la derivación de una o más representaciones de la estructura arquitectónica del sistema. Los estilos arquitectónicos alternativos o patrones son analizados con el fin de obtener la estructura que mejor se ajusta a los requisitos del cliente y a las normas de calidad. Una vez que es seleccionada una de las alternativas, la arquitectura se elabora utilizando el método de diseño arquitectónico.

¿Cuál es el producto obtenido? Durante el diseño arquitectónico se crea un modelo de arquitectura que abarca la arquitectura de datos y la estructura del programa. Además, se describen las propiedades de los componentes y sus relaciones (interacciones).

¿Cómo puedo estar seguro de que lo he hecho correctamente? En cada etapa, los productos resultantes del diseño del software son revisados para clarificar, corregir, completar y dar consistencia acorde a los requisitos establecidos, y entre unos y otros.

14.1 ARQUITECTURA DEL SOFTWARE

Shaw y Garlan [SHA96], en su libro de referencia sobre la materia, tratan la arquitectura del software de la siguiente forma:

> Incluso desde que el primer programa fue dividido en módulos, los sistemas de software han tenido arquitecturas, y los programadores han sido responsables de sus interacciones a través de módulos y de las propiedades globales de ensamblaje. Históricamente, las arquitecturas han estado implícitas —bien como accidentes en la implementación, bien como sistemas legados del pasado—. Los buenos desarrolladores de software han adoptado, a menudo, uno o varios patrones arquitectónicos como estrategias de organización del sistema, pero utilizaban estos patrones de modo informal y no tenían ningún interés en hacerlos explícitos en el sistema resultante.

Hoy, la arquitectura de software operativa y su representación y diseño explícitos se han convertido en temas dominantes de la ingeniería de software.

14.1.1. ¿Qué es arquitectura?

Cuando hablamos de la «arquitectura» de un edificio, nos vienen a la cabeza diferentes atributos. A nivel más básico, pensamos en la forma global de la estructura física. Pero, en realidad, la arquitectura es mucho más. Es la forma en la que los diferentes componentes del edificio se integran para formar un todo unido. Es la forma en la que el edificio encaja en su entorno y con los otros edificios de su vecindad. Es el grado en el que el edificio consigue su propósito fijado y satisface las necesidades de sus propietarios. Es el sentido estético de la estructura —el impacto visual del edificio— y el modo en el que las texturas, los colores y los materiales son combinados para crear la fachada externa y el «entorno vivo» interno. Son los pequeños detalles - el diseño de las instalaciones eléctricas, del tipo de suelo, de la colocación de tapices y una lista casi interminable—. Y, finalmente, es arte.

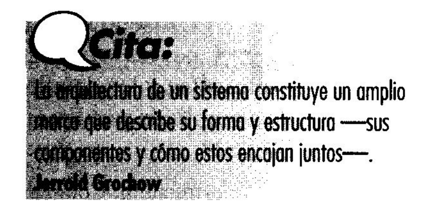


Se puede encontrar una lista útil de recursos de arquitectura de software en www2.umassd.edu/SECenter/SAResources.html

Pero, ¿qué pasa con la arquitectura de software? Bass y sus colegas [BAS98] definen este esquivo término de la siguiente forma:

> La arquitectura de software de un sistema de programa o computación es la estructura de las estructuras del sistema, la cual comprende los componentes del software, las propiedades de esos componentes visibles externamente, y las relaciones entre ellos.

La arquitectura no es el software operacional. Más bien, es la representación que capacita al ingeniero del software para: (1) analizar la efectividad del diseño para la consecución de los requisitos fijados, (2) considerar las alternativas arquitectónicas en una etapa en la cual hacer cambios en el diseño es relativamente fácil, y (3) reducir los riesgos asociados a la construcción del software.



La arquitectura de un sistema constituye un amplio marco que describe su forma y estructura —sus componentes y cómo estos encajan juntos—.
Jerrold Grochow

La definición presentada anteriormente enfatiza el papel de los «componentes del software» en cualquier representación arquitectónica. En el contexto del diseño arquitectónico, un componente del software puede ser tan simple como un módulo de programa, pero también puede ser algo tan complicado como incluir bases de datos y software intermedio («*middleware*») que permiten la configuración de una red de clientes y servidores. Las propiedades de los componentes son aquellas características necesarias para entender cómo los componentes interactúan con otros componentes. A nivel arquitectónico, no se especifican las propiedades internas (por ejemplo, detalles de un algoritmo). Las relaciones entre los componentes pueden ser tan sencillas como una llamada de procedimiento de un módulo a otro, o tan complicadas como el protocolo de acceso a bases de datos.

En este libro, el diseño de la arquitectura de software tiene en cuenta dos niveles de la pirámide del diseño (Fig. 13.1)—el diseño de datos y arquitectónico—. En este sentido, el diseño de datos nos facilita la representación de los componentes de datos de la arquitectura. El diseño arquitectónico se centra en la representación de la estructura de los componentes del software, sus propiedades e interacciones.

14.1.2. ¿Por qué es importante la arquitectura?

En su libro dedicado a la arquitectura de software, Bass y sus colegas [BAS98] identifican tres razones clave por las que la arquitectura de software es importante:

- las representaciones de la arquitectura de software facilitan la comunicación entre todas las partes (partícipes) interesadas en el desarrollo de un sistema basado en computadora;
- la arquitectura destaca decisiones tempranas de diseño que tendrán un profundo impacto en todo el trabajo de ingeniería del software que sigue, y es tan importante en el éxito final del sistema como una entidad operacional;
- la arquitectura «constituye un modelo relativamente pequeño e intelectualmente comprensible de cómo está estructurado el sistema y de cómo trabajan juntos sus componentes» [BAS98].



El modelo arquitectónico proporciona una visión «Gestalt» del sistema, permitiendo al ingeniero del software examinarlo como un todo.

El modelo de diseño arquitectónico y los patrones arquitectónicos contenidos dentro son transferibles. Esto es, los estilos y patrones de arquitectura (Sección **14.3.1)** pueden ser aplicados en el diseño de otros sistemas y representados a través de un conjunto de abstracciones que facilitan a los ingenieros del software la descripción de la arquitectura de un modo predecible.

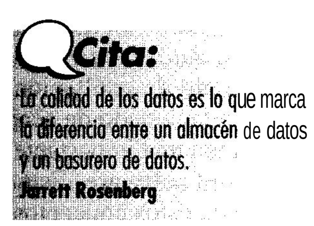
14.2 DISEÑO DE DATOS

Al igual que otras actividades de la ingeniería del software, el *diseño de datos* (a veces llamado *arquitectura de datos*) crea un modelo de datos y/o información que se representa con un alto nivel de abstracción (la visión de datos del cliente/usuario). Este modelo de datos, es entonces refinado en progresivas representaciones específicas de la implementación, que pueden ser procesadas por un sistema basado en computadora. En muchas aplicaciones de software, la arquitectura de datos tendrá una gran influencia sobre la arquitectura del software que debe procesarlo.

La estructura de datos ha sido siempre una parte importante del diseño de software. Al nivel de los componentes del programa, el diseño de las estructuras de datos y de los algoritmos asociados requeridos para su manipulación, son la parte esencial en la creación de aplicaciones de alta calidad. A nivel de aplicación, la traducción de un modelo de datos (derivado como parte de la ingeniería de requisitos) en una base de datos es el punto clave para alcanzar los objetivos de negocio del sistema. **A** nivel de negocios, el conjunto de información almacenada en las diferentes bases de datos y reorganizada en el almacén de datos facilita la minería de datos o el descubrimiento de conocimiento que puede influir en el propio éxito del negocio. De cualquier modo, el diseño de datos juega un papel muy importante.

14.2.1. Modelado de datos, estructuras de datos, bases de datos y almacén de datos

Los objetos de datos definidos durante el análisis de requisitos del software son modelados utilizando diagramas de entidad-relación y el diccionario de datos (Capítulo 12). La actividad de diseño de datos traduce esos elementos del modelo de requisitos en estructuras de datos a nivel de los componentes del software y, cuando es necesario, a arquitectura de base de datos a nivel de aplicación.



Durante muchos años, la arquitectura de datos estuvo limitada, generalmente, a las estructuras de datos a nivel del programa y a las bases de datos a nivel de aplicación. Pero hoy, las empresas grandes y pequeñas están inundadas de datos. No es inusual, incluso para una mediana empresa, tener docenas de bases de datos sirviendo diferentes aplicaciones de cientos de gigabytes de datos. El desafío de las empresas ha sido extraer información útil de su entorno de datos, particularmente cuando la información deseada está funcionalmente interrelacionada (por ejemplo, información que sólo puede obtenerse si los datos de marketing específicos se cruzan con los datos de la ingeniería del producto).

Para resolver este desafío, la comunidad de empresas de TI ha desarrollado las técnicas de *minería de datos,* también llamadas *descubrimiento de conocimiento en bases de datos (DCBC),*que navegan a través de las bases de datos en un intento por extraer el nivel de información de negocio apropiado. Sin embargo, la existencia de múltiples bases de datos, sus diferentes estructuras, el grado de detalle contenido en las bases de datos, y muchos otros factores hacen difícil la minería de datos dentro de un entorno con bases de datos existentes. Una solución alternativa, llamada almacén de datos, añade una capa adicional a la arquitectura de datos.



Para obtener información actualizada sobre tecnologías de almacén de datos visitar:
www.datawarehouse.com

Un *almacén de datos* es un entorno de datos separado, que no está directamente integrado con las aplicaciones del día a día, pero que abarca todos los datos utilizados por una empresa [MAT96]. En cierto sentido, un almacén de datos es una gran base de datos independiente, que contiene algunos, pero no todos los datos almacenados en las bases de datos que sirven al conjunto de aplicaciones requeridas en un negocio. Sin embargo, hay muchas características diferenciales entre un almacén de datos y una base de datos típica [INM95]:

Orientación por materia. Un almacén de datos está organizado por las materias importantes del negocio, más que por los procesos o funciones del negocio. Esto nos lleva a una exclusión de datos que podrían ser necesarios para una función particular del negocio, pero que generalmente no son necesarios para la minería de datos.

Integración. Sin tener en cuenta la fuente de datos, da consistencia nombrar convenciones, unidades y medidas, estructuras de codificación y atributos físicos incluso cuando la inconsistencia existe a través de las diferentes bases de datos orientadas a aplicaciones.

Restricciones de tiempo. Para un entorno de aplicación orientado a transacción, los datos son precisos en el momento del acceso y por un periodo de tiempo relativamente pequeño (normalmente de 60 a 90 días) antes del acceso. **Sin** embargo, en un almacén de datos, se accede a los datos en un momento específico del tiempo (por ejemplo, los clientes con los que se ha contactado el mismo día que el nuevo producto ha sido anunciado en la prensa comercial). El horizonte típico de tiempo de un almacén de datos es de *5* **a** 10 años.

No volatilidad. A diferencia de las típicas bases de datos de aplicaciones de negocios que atraviesan una continua corriente de cambios (insertar, borrar, actualizar), en el almacén de datos los datos se cargan, pero después de la primera transferencia, los datos no cambian.



Un almacén de datos contiene todos los datos utilizados en una empresa. Su objetivo es facilitar **el** acceso a ((conocimiento)) que de otro modo no se dispondría.

Estas características presentan un cuadro Único de desafíos de diseño para el arquitecto de datos.

14.2.2. Diseño de datos a nivel de componentes

El diseño de datos a nivel de componentes se centra en la representación de estructuras de datos a las que se accede directamente a través de uno o más componentes del software. Wasserman [WASSO] ha propuesto un conjunto de principios que pueden emplearse para especificar y diseñar dicha estructura de datos. En realidad, el diseño de datos empieza durante la creación del modelo de análisis. Recordando que el análisis de requisitos y el diseño a menudo se solapan, consideramos el siguiente conjunto de principios [WASSO] para la especificación de datos:

1. *Los principios del análisis sistemático aplicados a la función y al comportamiento deberían aplicarse también a los datos.* Invertimos mucho tiempo y esfuerzo en obtener, revisar y especificar los requisitos funcionales y el diseño preliminar. Las representaciones de flujo de datos y contenido deberían desarrollarse y revisarse, las de objetos de datos deberían identificarse, se deberían estudiar las organizaciones alternativas de datos y debería evaluarse el impacto del modelado de datos sobre el diseño del software. Por ejemplo, la especificación de una lista enlazada con múltiples anillos puede satisfacer los requisitos de los datos pero puede llevar a un diseño del software **poco** flexible. Una organización de datos alternativa nos podría llevar a obtener mejores resultados.



2. *Todas las estructuras de datos y las operaciones a llevar a cabo en cada una de ellas deberían estar claramente identificadas.* El diseño de una estructura de datos eficaz debe tener en cuenta las operaciones que se van a llevar a cabo sobre dicha estructura de datos (vea [AHO83]). Por ejemplo, imagínese una estructura de datos hecha con un conjunto de elementos de datos diversos. La estructura de datos se va a manipular con varias funciones principales del software. **Des**pués de la evaluación de la operación realizada sobre la estructura de datos, se define un tipo de datos abstracto para usarlo en el diseño del software subsiguiente. La especificación de los tipos de datos abstractos puede simplificar considerablemente el diseño del software.
3. *Se debería establecer un diccionario de datos y usarlo para definir el diseño de los datos y del* ***programa.*** El concepto de diccionario de datos se introdujo en el Capítulo 12. Un diccionario de datos representa explícitamente las relaciones entre los objetos de datos y las restricciones de los elementos de una estructura de datos. Los algoritmos que deben aprovecharse de estas relaciones específicas pueden definirse más fácilmente si existe una especificación de datos tipo diccionario.
4. *Las decisiones de diseño de datos de bajo nivel deberían dejarse para el final del proceso de diseño.* Se puede usar un proceso de refinamiento paso a paso para el diseño de datos. Es decir, la organización general de datos puede definirse durante el análisis de requisitos; refinarse durante los trabajos de diseño de datos **y** especificarse en detalle durante el diseño a nivel de componentes. El enfoque descendente del diseño de **data** proporciona ventajas análogas al enfoque descendente del diseño de software: se diseñan y evalúan primeramente los atributos estructurales principales de manera que se pueda establecer la arquitectura de los datos.
5. *La representación de las estructuras de datos deberían conocerla sólo aquellos módulos que deban hacer uso directo de los datos contenidos dentro de la estructura.* El concepto de ocultación de información y el concepto relacionado de acoplamiento (Capítulo 13) proporciona una importante visión de la calidad del diseño del software. Este principio alude a la importancia de estos conceptos así como

a «la importancia de separar la visión lógica de un objeto de datos de su visión física» [WASSO].

6. *Debería desarrollarse una biblioteca de estructuras de datos útiles y de las operaciones que se les pueden aplicar.* Las estructuras de datos y las operaciones deberían considerarse como recursos para el diseño del software. Las estructuras de datos pueden diseñarse para que se puedan reutilizar. Una biblioteca de plantillas de estructuras de datos (tipos abstractos de datos) puede reducir el esfuerzo del diseño y de la especificación de datos.

7. *Un diseño del software y un lenguaje de programación debería soportar la especificación y realización de los tipos abstractos de datos.* La implementación de una estructura de datos sofisticada puede hacerse excesivamente difícil si no existen los medios de especificación directa de la estructura en el lenguaje de programación escogido para dicha implementación.

Los principios descritos anteriormente forman una base para un enfoque de diseño de datos a nivel de componentes que puede integrarse en las fases de análisis y diseño.

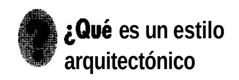
14.3 ESTILOS ARQUITECTÓNICOS

Cuando un constructor utiliza el término *«centre hall colonial»* para describir una casa, la mayoría de la gente familiarizada con las casas en los Estados Unidos sería capaz de recrear una imagen general de cómo sería esa casa y de cómo sena su diseño de plantas. El constructor ha utilizado un *estilo arquitectónico* a modo de mecanismo descriptivo para diferenciar esa casa de otros estilos (por ejemplo, *A-ame, raised ranch, Cape Cod*). Pero, lo más importante, es que el estilo arquitectónico es también un patrón de construcción. **Más** adelante se deberán definir los detalles de la casa, se especificarán sus dimensiones finales, se le añadirán características personalizadas y se **determinarán** los materiales de construcción, pero el patrón --*«centre hall colonial»*-- guía al constructor en su trabajo.



El proyecto ABLE de lo CMU cuenta con trabajos y ejemplos muy útiles sobre estilos arquitectónicos:
tom.cs.cmu.edu/able/

El software construido para sistemas basados en computadoras también cuenta con diversos estilos arquitectónicos[1]. Cada estilo describe una categoría del sistema que contiene: (1) un conjunto de *componentes* (por ejemplo, una base de datos, módulos computacionales) que realizan una función requerida por el sistema; (2) un conjunto de *conectores* que posibilitan la «comunicación, la coordinación y la cooperación» entre los componentes; (3) *restricciones* que definen cómo se pueden integrar los componentes que forman el sistema; y (4) *modelos semánticos* que permiten al diseñador entender las propiedades globales de un sistema para analizar las propiedades conocidas de **sus** partes constituyentes [BAS98]. En la siguiente sección, abordamos los patrones arquitectónicos comúnmente utilizados para el software.



14.3.1. Una breve taxonomía de estilos y patrones

Aunque durante los pasados 50 años se han creado cientos de miles de sistemas basados en computadora, la gran mayoría pueden ser clasificados (ver [SHA96], [BAS98], [BUS98]) dentro de uno de los estilos arquitectónicos:

Arquitecturas centradas de datos. En el centro de esta arquitectura se encuentra un almacén de datos (por ejemplo, un documento o una base de datos) al que otros componentes acceden con frecuencia para actualizar, añadir, borrar o bien modificar los datos del almacén. La figura **14.1** representa un estilo típico basada en los datos. El software de cliente accede a un almacén centrai. En algunos casos, el almacén de datos es *pasivo.* Esto significa que el software de cliente accede a los datos independientemente de cualquier cambio en los datos o de las acciones de otro software de cliente. Una variación en este acceso transforma el almacén en una «pizarra» que envía notificaciones al software de cliente cuando los datos de interés del cliente cambian.

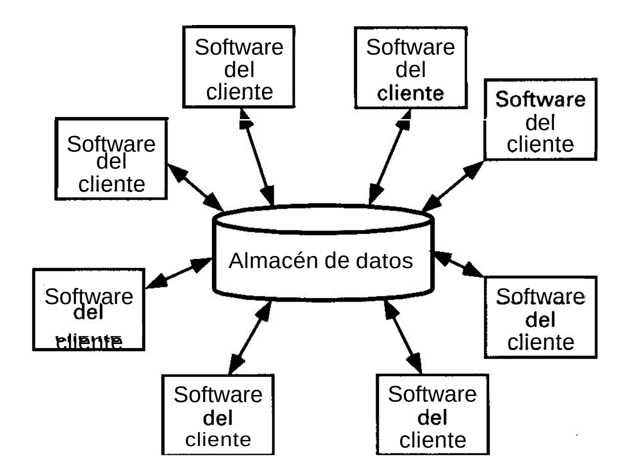


FIGURA 14.1. Arquitectura basada en los datos.

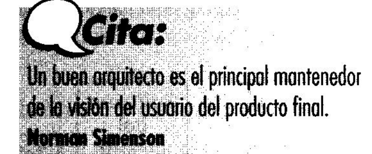
[1] Los términos *estilos* y *patrones* se utilizan indistintamente en este capítulo.

Las arquitecturas basadas en **los** datos promueven la capacidad de integración (integrability) [BAS98]. Por consiguiente, **los** componentes existentes pueden cambiarse o **los** componentes del nuevo cliente pueden añadirse a la arquitectura sin involucrar a otros clientes (porque **los** componentes del cliente operan independientemente). Además, **los** datos pueden ser transferidos entre **los** clientes utilizando un mecanismo de pizarra (por ejemplo, el componente de pizarra sirve para coordinar la transferencia de información entre clientes). Los componentes cliente son procesos ejecutados independientemente.

Cita:
El uso de patrones y estilos de diseño está impregnado en las disciplinas de ingeniería.
Mary Shaw y David Garlan

Arquitecturas de flujo de datos. Esta arquitectura se aplica cuando **los** datos de entrada son transformados a través de una serie de componentes *computacionales* o *manipulativos* en los datos de salida. Un patrón tubería y filtro (Fig. 14.2.a) tiene un grupo de componentes, llamados filtros, conectados por tuberías que transmiten datos de un componente al siguiente. Cada filtro trabaja independientemente de aquellos componentes que se encuentran en el flujo de entrada o de salida; está diseñado para recibir la entrada de datos de una cierta forma y producir una salida de datos (hacia el siguiente filtro) de una forma específica. Sin embargo, el filtro no necesita conocer el trabajo de **los** filtros vecinos.

Si el flujo de datos degenera en una simple línea de transformadores (Fig. 14.2.b) se le denomina *secuencia1 por lotes*. Este patrón aplica una serie de componentes secuenciales (filtros) para transformarlos.



Arquitecturas de llamada y retorno. Este estilo arquitectónico permite al diseñador del software (arquitecto del sistema) construir una estructura de programa relativamente fácil de modificar y ajustar a escala. Existen dos subestilos [BAS98] dentro de esta categoría:

- *arquitecturas de programa principallsuhprograma:* esta estructura clásica de programación descompone las funciones en una jerarquía de control donde un programa «principal» llama a un número de componentes del programa, **los** cuales, en respuesta, pueden también llamar a otros componentes. La Figura 13.3 representa una arquitectura de este tipo.
- *arquitecturas de llamada de procedimiento remoto:* los componentes de una arquitectura de programa principal/subprograma, están distribuidos entre varias computadoras en una red.

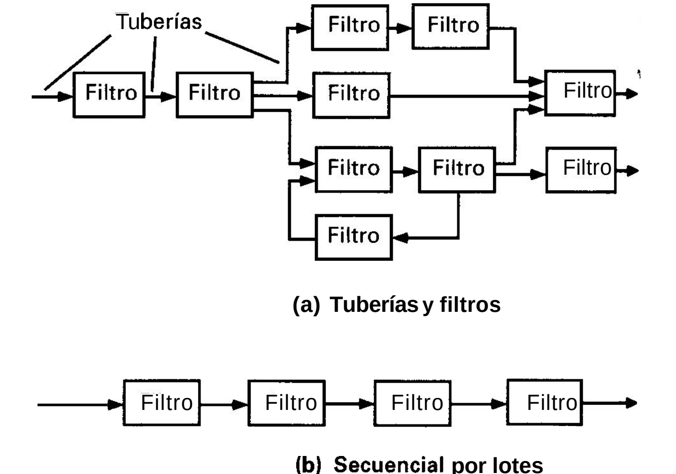


FIGURA 14.2. Arquitecturas de flujo de datos.

Referencia cruzada
En la Parte 4 se presenta un estudio detallado sobre las arquitecturas orientadas a objetos.

Arquitecturas orientadas a objetos. Los componentes de un sistema encapsulan **los** datos y las operaciones que se deben realizar para manipular **los** datos. La comunicación y la coordinación entre componentes se consigue a través del paso de mensajes.

Arquitecturas estratificadas. La estructura básica de una arquitectura estratificada se representa en la Figura 14.3. Se crean diferentes capas y cada una realiza operaciones que progresivamente se aproximan más al cuadro de instrucciones de la máquina. En la capa externa, **los** componentes sirven a las operaciones de interfaz de usuario. En la capa interna, **los** componentes realizan operaciones de interfaz del sistema. Las capas intermedias proporcionan servicios de utilidad y funciones del software de aplicaciones.

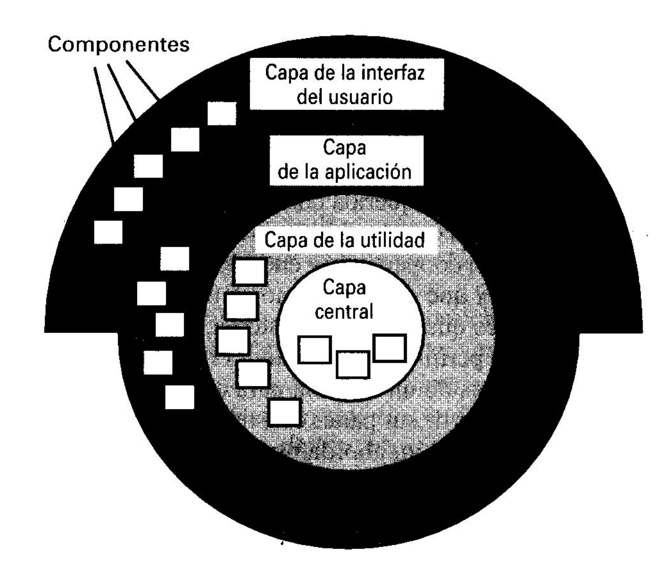


FIGURA 14.3. Arquitectura ectratificada.

Los estilos arquitectónicos citados anteriormente son sólo una pequeña parte de los que dispone el diseñador de software[2]. Una vez que la ingeniería de requisitos define las características y las restricciones del sistema que ha de ser construido, se escoge el patrón arquitectónico (estilo) o la combinación de patrones (estilos) que mejor encajan con las características y restricciones. En muchos casos, puede ser apropiado más de un patrón y se podrían diseñar y evaluar estilos arquitectónicos alternativos.

14.3.2. Organización y refinamiento

Puesto que el proceso de diseño deja a menudo al ingeniero con un número de alternativas arquitectónicas, es importante establecer un conjunto de criterios de diseño que puedan ser utilizados para evaluar un diseño arquitectónico derivado. Las siguientes cuestiones [BAS98] proporcionan una idea del estilo arquitectónico que ha sido derivado:

Control. ¿Cómo se gestiona el control dentro de la arquitectura? ¿Existe una jerarquía de control diferente, y si es así, cuál es el papel de los componentes dentro de esa jerarquía de control? ¿Cómo transfieren el control los componentes dentro del sistema? ¿Cómo se comparte el control entre componentes? ¿Cuál es la topología de control (por ejemplo, la forma geométrica[3] que adopta el control)?, ¿Está el control sincronizado o los componentes actúan asincrónicamente?

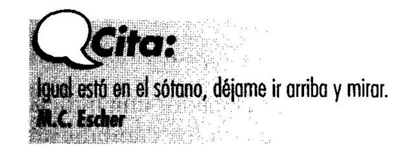


Datos. ¿Cómo se comunican los datos entre componentes? ¿El flujo de datos es continuo o son objetos de datos que pasan de un componente a otro, o los datos están disponibles globalmente para ser compartidos entre los componentes del sistema? ¿Existen los componentes de datos (por ejemplo, una pizarra o almacén), y si es así, cuál es su papel? ¿Cómo interactúan los componentes funcionales con los componentes de datos? ¿Los componentes de datos son activos o pasivos (por ejemplo, los componentes de datos interactúan activamente con otros componentes del sistema)? ¿Cómo interactúan los datos y el control dentro del sistema?

Estas preguntas proporcionan al diseñador una evaluación temprana de la calidad del diseño y sientan las bases para un análisis más detallado de la arquitectura.

14.4 ANÁLISIS DE DISEÑOS ARQUITECTÓNICOS ALTERNATIVOS

Las preguntas citadas en la sección anterior proporcionan una evaluación preliminar del estilo arquitectónico escogido para un sistema concreto. Sin embargo, para la consecución efectiva del diseño, se necesita un método de evaluación de la calidad de la arquitectura más completo. En las siguientes secciones, consideramos dos enfoques diferentes para el análisis de diseños arquitectónicos alternativos. El primer enfoque utiliza un método iterativo para evaluar el diseño de los compromisos. El segundo enfoque aplica una pseudotécnica cuantitativa para evaluar la calidad del diseño.



14.4.1. Un método de análisis de compromiso para la arquitectura

El *Instituto de Ingeniería de Software (SEI)* ha desarrollado un *Método de análisis de compromiso para la arquitectura* (MACA)[KAZ98] que establece un proceso de evaluación iterativo para las arquitecturas de software. Las actividades de análisis de diseño mencionadas abajo se realizan iterativamente:

1. *Recopilación de escenarios.* En el Capítulo 11 se recogen un grupo de casos de uso para representar el sistema desde el punto de vista del usuario.
2. *Elicitación de requisitos.* Esta información es requerida como parte de los requisitos de ingeniería y se utiliza para asegurarse de que todos los clientes, usuarios y partícipes implicados han sido atendidos.



Se puede encontrar información detallada sobre análisis de compromiso de software arquitectónico en: **www.sei.cmu.edu/ata/ata_method.html**

3. *Describir los patrones/estilos arquitectónicos escogidos para derivar los escenarios y requisitos.* El(los) estilo(s) debería describirse utilizando vistas arquitectónicas como:
   - *vista de módulo:* para analizar el trabajo asignado por componente y el grado de ocultación de información que se ha alcanzado
   - *vista de proceso:* para analizar la actuación del sistema
   - *vista de flujo de datos:* para analizar el grado en el que la arquitectura cumple los requisitos funcionales

Referencia cruzada

En los Capítulos 8 y 19 se estudian los atributos de calidad.

[2] Ver [SHA97], [BAS98] y [BUS98] para un estudio detallado de estilos y patrones arquitectónicos.

[3] Una jerarquía es una forma geométrica, pero también podemos encontrar mecanismos de control semejantes en un sistema cliente/servidor.

4. *Evaluar los atributos de calidad considerando cada atributo de forma aislada.* El número de atributos de calidad escogidos para el análisis depende del tiempo disponible para la revisión y del grado de relevancia de dichos atributos para el sistema actual. Los atributos de calidad para la evaluación del diseño arquitectónico incluyen: fiabilidad, rendimiento, seguridad, facilidad de mantenimiento, flexibilidad, capacidad de prueba, movilidad, reutilización e interoperatibilidad.

5. *Identificar la sensibilidad de los atributos de calidad con los diferentes atributos arquitectónicos en un estilo arquitectónico específico.* Esto se puede conseguir realizando pequeños cambios en la arquitectura y determinando cuán sensibles al cambio son los atributos de calidad, como el rendimiento. Aquellos atributos afectados significativamente por un cambio en la arquitectura son denominados *puntos sensibles.*
6. *Análisis de las arquitecturas candidatas (desarrolladas en el paso 3) utilizando el análisis de sensibilidad del paso 5.* El SEI describe este enfoque de la siguiente forma [KAZ98]:

> Una vez determinados los puntos sensibles de la arquitectura, encontrar los puntos de compromiso consiste simplemente en la identificación de los elementos arquitectónicos en los cuales **varios** atributos son sensibles. **Por** ejemplo, el rendimiento de una arquitectura cliente/servidor sena muy sensible al número de servidores (el rendimiento aumenta, dentro de un grado, aumentando en número de servidores). La disponibilidad de esa arquitectura también variaría directamente con el número de servidores. Sin embargo, la seguridad del sistema variaría inversamente al número de servidores (porque el sistema contiene más puntos de ataque potenciales). El número de servidores, entonces, es un punto de compromiso con respecto a la arquitectura. Este es **un** elemento, potencialmente uno de muchos, donde se harán los compromisos arquitectónicos, consciente o inconscientemente.

Estos seis pasos representan la primera iteración MACA. Tras los resultados de los pasos 5 y 6 algunas arquitecturas alternativas se eliminarían, una o varias de las arquitecturas restantes serían modificadas y presentadas con mayor detalle, y después los pasos MACA se aplicarían de nuevo.

14.4.2. Guía cuantitativa para el diseño arquitectónico

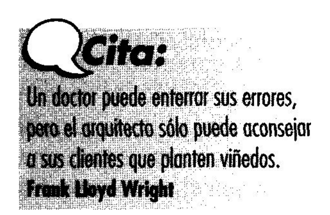
Uno de los muchos problemas con los que se enfrentan los ingenieros del software durante el proceso de diseño es la carencia general de métodos cuantitativos para la evaluación de la calidad de los diseños propuestos. El enfoque MACA recogido en la Sección 14.4.1 representa un enfoque innegablemente cualitativo y útil para el análisis del diseño.

Asada [SHA96] propone varios modelos simples **para** ayudar al diseñador a determinar el grado al cual una arquitectura particular alcanza los criterios de «bondad» predefinidos. Estos criterios, a veces llamados *dimensiones del diseño,* a menudo abarcan los atributos de calidad definidos en la sección anterior: fiabilidad, **rendimiento**, seguridad, facilidad de mantenimiento, flexibilidad, capacidad de prueba, movilidad, reutilización e interoperatibilidad, entre otros.

El primer modelo, denominado *análisis del espectro,* evalúa un diseño arquitectónico en un espectro de «bondad» desde el mejor diseño posible hasta el peor. Una **vez** que la arquitectura del software ha sido propuesta, se analiza asignando una «puntuación» a cada una de las dimensiones del diseño. Estas notas de las dimensiones se suman para determinar la calificación total, S, del diseño como un todo. Los casos de notas más bajas[4] son asignados a un diseño hipotético, y se computa la **suma** de notas de la peor arquitectura, S_w. La mejor nota, S_b, se computa para un diseño óptimo[5]. Entonces calculamos el *índice del espectro,* I_s, mediante la ecuación:

$$I_s = [(S - S_w)/(S_b - S_w)] \times 100$$

El índice del espectro indica el grado al cual la arquitectura propuesta se aproxima al sistema óptimo dentro del espectro de alternativas razonables para el diseño.



Si se realizan modificaciones del diseño propuesto o si se propone un nuevo diseño entero, los índices de espectro de ambos deben ser comparados y se computará un *índice de mejora,* I_{me}:

$$I_{me} = I_{s1} - I_{s2}$$

Esto proporciona al diseñador una indicación relativa a la mejora asociada con los cambios arquitectónicos realizados o con la nueva arquitectura propuesta. Si I_{me} es positivo, entonces podemos concluir que el sistema 1 ha sido mejorado en relación al sistema 2.

El *análisis de selección del diseño* es otro modelo que requiere un cuadro de dimensiones de diseño **para** ser definido. La arquitectura propuesta es entonces evaluada para determinar el número de dimensiones del diseño que se logran cuando se compara con un sistema ideal (el mejor caso). Por ejemplo, si una arquitectura propuesta alcanzara una reutilización de componentes excelente, **y** esta dimensión es requerida para el sistema ideal, la dimensión de reutilización ha

[4] El diseño debe ser todavía aplicable al problema actual, incluso si la solución no es particularmente buena.

[5] El diseño seria Óptimo, pero las restricciones, los costes u otros factores no permitirán su construcción.

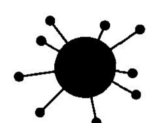
sido lograda. Si la arquitectura propuesta tiene poca seguridad, y se necesita una gran seguridad, no se ha alcanzado la dimensión del diseño.

Calculamos el *índice de selección del diseño, d,* como:

$$d=(N_s/N_a)\times 100$$

donde N_s es el número de dimensiones de diseño logradas en la arquitectura propuesta y N_a es el número total de dimensiones en el espacio de diseño. A mayor índice de selección del diseño, más cerca estamos de que la arquitectura propuesta alcance el sistema ideal.

El *análisis de contribución* «identifica las razones por las que un grupo de alternativas de diseño obtiene unas notas menores que otro». [SHA96] Retomando el estudio del *despliegue de la función de calidad (DFC)* visto en el Capítulo 11, el análisis de valor se realiza para determinar la prioridad relativa de los requisitos determinados durante el despliegue de funciones, el despliegue de información y el despliegue de tareas. Se identifica un conjunto de «mecanismos de realización» (características de la arquitectura). Se crea un listado con todos los requisitos del cliente (determinados a través del DFC) y una matriz de referencias cruzadas. Las celdas de la matriz indican (en una escala del 1 al 10) la fuerza relativa de la relación entre un mecanismo de realización y un requisito para cada arquitectura propuesta. A esto se le conoce como *espacio de diseño cuantificado (EDC).*



Hoja de cálculo DFC

El EDC es bastante fácil de implementar como un modelo de hoja de cálculo y puede ser utilizado para aislar porqué un cuadro de alternativas de diseño obtiene unas notas menores que otro.

14.4.3. Complejidad arquitectónica

Para evaluar la complejidad total de una arquitectura dada, una técnica útil consiste en considerar las relaciones de dependencia entre los componentes de la arquitectura. Estas relaciones de dependencia son conducidas a través de los flujos de información/control dentro del sistema.

Zhao [ZHA98] propone tres tipos de dependencia:

Dependencias de compartimiento, representan las relaciones de dependencia entre los consumidores que utilizan los mismos recursos o los productores que producen para los mismos consumidores. Por ejemplo, tenemos dos componentes u y v, si *u* y v se refieren a los mismos datos globales, entonces existe una dependencia de compartimiento entre ambos.

Dependencias de flujo, representan *las* relaciones de dependencia entre los productores y los consumidores de recursos. Por ejemplo, entre dos componentes u y v, si u debe completarse antes de que el control fluya a *v* (prerrequisito), o *si* u y *v* se comunican a través de parámetros, entonces existirá una relación de dependencia de flujo entre ambos.

Dependencias restrictivas, representan las restricciones de un relativo flujo de control entre un cuadro de actividades. Por ejemplo, dos componentes u y v, si u *y v* no se pueden ejecutar al mismo tiempo (por exclusión mutua), entonces existirá una dependencia restrictiva entre u y *v.*

Las dependencias de compartimiento y de flujo recogidas por Zhao se parecen de alguna forma al concepto de acoplamiento visto en el Capítulo 13. En el Capítulo 19 se explicarán mediciones sencillas para la evaluación de las dependencias.

14.5 CONVERSIÓN DE LOS REQUISITOS EN UNA ARQUITECTURA DEL SOFTWARE

En el Capítulo 13 ya dijimos que los requisitos del software pueden ser convertidos en varias representaciones del modelo de diseño. Los estilos arquitectónicos vistos en la Sección 14.3.1 representan arquitecturas radicalmente diferentes, por lo cual no debería sorprendernos que no exista una única conversión que logre una transición del modelo de requisitos al modelo de diseño. De hecho, todavía no existe una conversión para algunos modelos arquitectónicos, y el diseñador debe lograr la traducción de los requisitos del diseño para esos estilos de una forma *ad hoc.*

Para ilustrar un enfoque de conversión arquitectónica, consideraremos la arquitectura de llamada y retorno —una estructura muy común para diferentes tipos de sistemas[6]—. La técnica de conversión que se presenta capacita al diseñador para derivar arquitecturas de llamada y retorno razonablemente complejas en diagramas de flujo de datos dentro del modelo de requisitos.

La técnica, a veces denominada *diseño estructurado,* tiene sus orígenes en antiguos conceptos de diseño que defendían la modularidad [DEN73], el diseño descendente [WIR71] y la programación estructurada

[6] También es importante mencionar que las arquitecturas de llamada y retorno pueden residir dentro de otras arquitecturas mas sofisticadas vistas anteriormente en este capitulo. Por ejemplo, la arquitectura de uno o vanos componentes de una arquitectura cliente/servidor podría ser de llamada y retorno.

[DAH72, LIN79]. Stevens, Myers y Constantine [STE74] propusieron el diseño del software basado en el fluyo de datos a través de un sistema. Los primeros trabajos se refinaron y se presentaron en libros de Myers [MYE78], Yourdon y Constantine [YOU79].

¿Cuáles son los pasos a seguir en la evaluación del DFD en una arquitectura de llamada y retorno?

El diseño estructurado suele caracterizarse como un método de diseño orientado al flujo de datos porque permite una cómoda transición desde el *diagrama de flujo de datos (DFD)* a la arquitectura de software[7]. La transición desde el flujo de información (representado como un diagrama de flujo de datos) a una estructura del programa se realiza en un proceso de seis pasos: (1) se establece el tipo de flujo de información; (2) se indican los límites del flujo; **(3)** se convierte el DFD en la estructura del programa; (4) se define la jerarquía de control; *(5)* se refina la estructura resultante usando medidas y heur'sticas de diseño, y (6) se refina y elabora la descripción arquitectónica.

El tipo de flujo de información es lo que determina el método de conversión requerido en el paso **3.** En las siguientes secciones examinamos los dos tipos de flujo.

14.5.1. Flujo de transformación

Retomando el modelo fundamental de sistema (nivel 0 del diagrama de flujo de datos), la información debe introducirse y obtenerse del software en forma de «mundo exterior». Por ejemplo, los datos escritos con un teclado, los tonos en una línea telefónica, y las imágenes de vídeo en una aplicación multimedia son todas formas de información del mundo exterior. Tales datos internos deben convertirse a un formato interno para el procesamiento. La información entra en el sistema a lo largo de caminos que transforman los datos externos a un formato interno. Estos caminos se identifican como flujo *de entrada.* En el interior del software se produce una transición. La información entrante se pasa a través de un *centro de transformación* y empieza a moverse a lo largo de caminos que ahora conducen hacia «fuera» del software. Los datos que se mueven a lo largo de estos caminos se denominan *flujo de salida.* **El** flujo general de datos ocurre de manera secuencial y sigue un, o unos pocos, caminos[8] «directos». Cuando un segmento de un diagrama de flujo de datos presenta estas características, lo que tenemos presente es un *flujo de transformación.*

Referencia cruzada
Los diagramas de **flujo** de datos se estudian en detalle en el Capítulo 12.

14.5.2. Flujo de transacción

El modelo fundamental de sistema implica un flujo de transformación; por tanto, es posible caracterizar todo el flujo de datos en esta categoría. Sin embargo, el flujo de información está caracterizado a menudo por un único elemento de datos, denominado *transacción,* que desencadena otros flujos de información a lo largo de uno de los muchos caminos posibles. Cuando un **DFD** toma la forma mostrada en la Figura 14.4, lo que tenemos es un *flujo de transacción.*

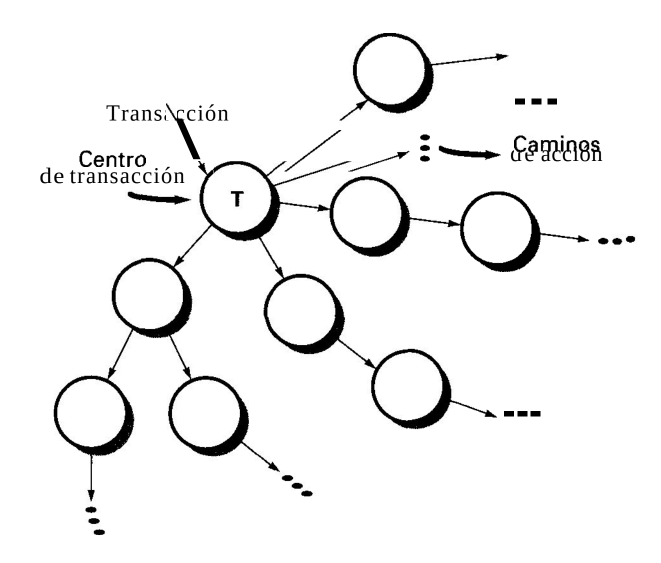


FIGURA 14.4. Flujo de transacción.

El flujo de transacción se caracteriza por datos que se mueven a lo largo de un camino de entrada que convierte la información del mundo exterior en una transacción. La transacción se evalúa y, basándose en ese valor, se inicia el flujo a lo largo de uno de muchos *caminos de acción.* El centro del flujo de información del que parten muchos de los caminos de acción se denomina *centro de transacción.*

Debería recalcarse que dentro del DFD de un sistema grande, ambos flujos de transacción y de transformación pueden presentarse. Por ejemplo, en un flujo orientado a transacción, el flujo de información a lo largo de un camino de acción puede tener características de flujo de transformación.

[7] Debe recordarse que también durante el modelado de análisis se utilizan otros elementos del método de análisis (por ejemplo; el diccionario de datos, EP, EC).

[8] Un análisis obvio de este tipo de flujo de información lo encontramos en la Sección 14.3.1 en la arquitectura de flujo de datos. Sin embargo, hay muchos casos en los que la arquitectura de flujo de datos no sería la mejor elección para un sistema complejo. Los ejemplos incluyen sistemas que sufrirían cambios sustanciales en el tiempo, o sistemas en los cuales el procesamiento asociado con el flujo de datos no es necesariamente secuencial.

14.6 ANÁLISIS DE LAS TRANSFORMACIONES

El *análisis de las transformaciones* es un conjunto de pasos de diseño que permite convertir un DFD, con características de flujo de transformación, en un estilo arquitectónico específico. En esta sección se describe el análisis de las transformaciones aplicando los pasos de diseño a un sistema de, por ejemplo, una parte del software de *Hogarseguro* presentado en capítulos anteriores.

14.6.1. Un ejemplo

El sistema de seguridad *Hogarseguro*, presentado anteriormente en este libro, es representativo de muchos productos y sistemas basados en computadora actualmente en uso. El producto vigila el mundo real y reacciona ante cambios que encuentra. También interacciona con el usuario a través de una serie de entradas por teclado y visualizaciones alfanuméricas. El nivel 0 del diagrama de flujo de datos de *Hogarseguro*, reproducido del Capítulo 12, se muestra en la Figura 14.5.



FIGURA 14.5. DFD a nivel de contexto **para** *Hogarseguro.*

Durante el análisis de requisitos, se habrán creado más modelos detallados del flujo para *Hogarseguro*. Además, se crearán las especificaciones de control y proceso, un diccionario de datos y varios modelos de comportamiento.

14.6.2. Pasos del diseño

El ejemplo anterior se usará para ilustrar cada paso en el análisis de las transformaciones. Los pasos empiezan con una reevaluación del trabajo hecho durante el análisis de requisitos y después evoluciona hacia el diseño de la arquitectura del software.

Paso 1. Revisar el modelo fundamental del sistema. El modelo fundamental del sistema comprende el DFD de nivel 0 y la información que lo soporta. En realidad, el paso de diseño empieza con una evaluación de la *especificación del sistema* y de la *especificación de requisitos del software*. Ambos documentos describen el flujo y la estructura de la información en la interfaz del software. Las Figuras 14.5 y 14.6 muestran el nivel 0 y 1 del flujo de datos del software *Hogarseguro*.

Paso 2. Revisar y refinar los diagramas de flujo de datos del software. La información obtenida de los modelos de análisis contenidos en la *Especificación de Requisitos del Software* se refina para obtener mayor detalle. Por ejemplo, se examina el DFD del nivel 2 de *monitorizar sensores* (Fig. 14.7) y se obtiene un diagrama de flujo de datos de nivel 3 como se muestra en la Figura 14.8. En el nivel 3, cada transformación en el diagrama de flujo de datos presenta una cohesión relativamente alta (Capítulo 13). Es decir, el proceso implicado por una transformación realiza una función única y distinta que puede implementarse como un módulo[9] en el software *HogarSeguro*. Por tanto, el DFD de la Figura 14.8 contiene suficiente detalle para establecer un diseño «inicial» de la arquitectura del subsistema de monitorizar sensores y continuamos sin más refinamiento.



Si el *DFD* es ***refinado más*** de uno vez, se ***esforzará*** por ***derivar burbujas*** que ***presentan gran*** cohesión.

Paso 3. Determinar si el DFD tiene características de flujo de transformación o de transacción. En general, el flujo de información dentro de un sistema puede representarse siempre como una transformación. Sin embargo, cuando se encuentra una característica obvia de transacción (Fig. 14.4), se recomienda una estructura de diseño diferente. En este paso, el diseñador selecciona la característica general del flujo (de la amplitud del software) basándose en la propia naturaleza del DFD. Además, se aíslan regiones locales de flujo de transformación o de transacción. Estos *subflujos* pueden usarse para refinar la estructura del programa obtenida por la característica general que prevalece. Por ahora, concentraremos nuestra atención solamente en el flujo de datos del subsistema de monitorización de sensores mostrado en la Figura 14.7.



A menudo se podrán encontrar ambos tipos de **flujo** de datos dentro del mismo modelo de análisis. tos **flujos** se dividen y **la** estructura **del** programa se deriva utilizando el análisis adecuado.

9 La utilización del término módulo en este capítulo equivale al término componente usado anteriormente en el estudio de la arquitectura de software.

Evaluando el DFD (Fig. 14.8), vemos que los datos entran al software por un camino de entrada y salen por tres caminos de salida. No se distingue ningún centro de transacción (aunque la transformación *establecer las condiciones de alarma* podría percibirse como tal). Por tanto, se asumirá una característica general de transformación para el flujo de información.

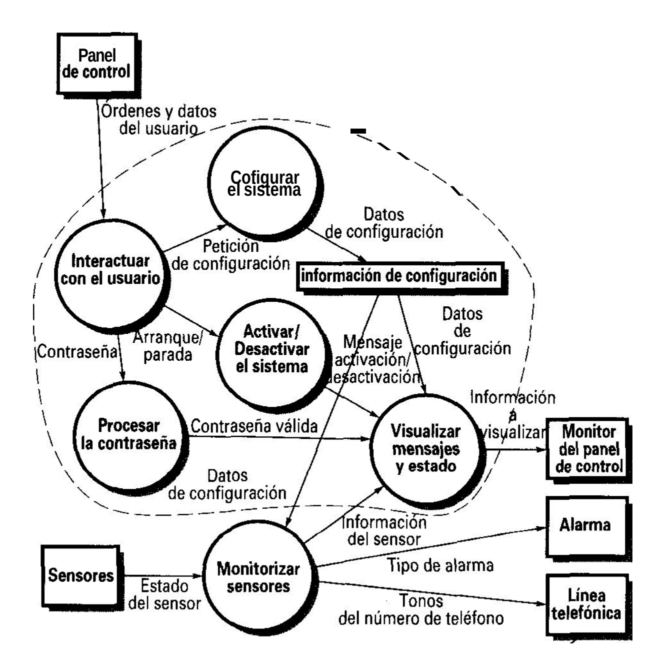


FIGURA 14.6. Nivel 1 del DFD del *HogarSeguro.*

Paso 4. Aislar el centro de transformación especificando los límites de los flujos de entrada y salida. En la sección precedente el flujo de entrada se describió como un camino en el que la información se convierte de formato externo en interno; el flujo de salida convierte de formato interno a externo. Los límites del flujo de entrada y el de salida son interpretables. Es decir, los diseñadores pueden elegir puntos ligeramente diferentes como límites de flujo. De hecho, se pueden obtener soluciones de diseño alternativas variando la posición de los límites de flujo. Aunque hay que tener cuidado cuando se seleccionan los límites, una variación de una burbuja a lo largo de un camino de flujo tendrá generalmente poco impacto en la estructura final del programa.



Cambia lo situación de los fronteras de flujo en un esfuerzo por explorar los estructuras de programa alternativas. No llevo mucho tiempo y puede proporcionarnos importantes ideas.

Los límites de flujo del ejemplo se ilustran como curvas sombreadas que van verticales a través del flujo en la Figura 14.8. Las transformaciones (burbujas) que forman el centro de transformación se encuentran entre los dos límites sombreados que van de arriba abajo en el dibujo. Se podría argumentar algún cambio para reajustar los límites (por ejemplo, se podría proponer un límite del flujo de entrada que separe *leer sensores,* de *adquirir información de respuesta).* El énfasis en este paso del diseño debería ponerse en seleccionar límites razonables, en vez de largas disquisiciones sobre la colocación de las separaciones.

Paso 5. Realizar una «descomposición de primer nivel». La estructura del programa representa una distribución descendente del control. La descomposición en partes provoca una estructura de programa en la que los módulos del nivel superior realizan la toma de decisiones y los módulos del nivel inferior realizan la mayoría del trabajo de entrada, cálculos y salida. Los módulos de nivel intermedio realizan algún control y cantidades moderadas de trabajo.

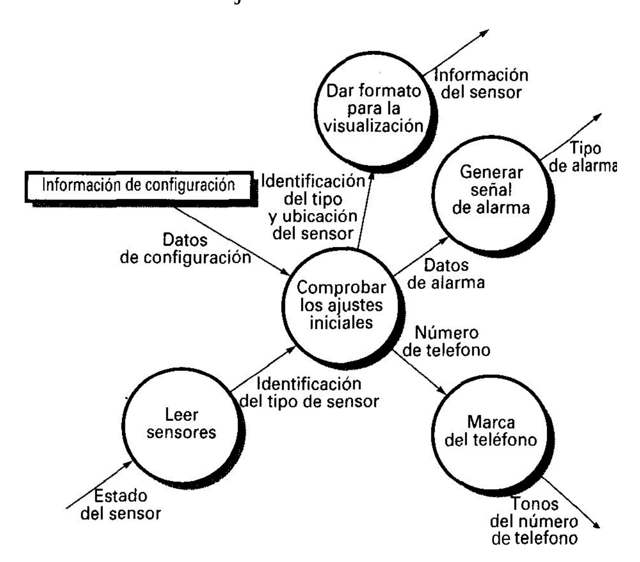


FIGURA 14.7. Nivel 2 del DFD que refina el proceso de monitorizar sensores

Cuando encontramos un flujo de transformación, un DFD se organiza en una estructura específica (una arquitectura de llamada y retorno) que proporciona control para el procesamiento de información de entrada, transformación y de la salida. Esta descomposición en factores o primer nivel del subsistema de monitorizar sensores se ilustra en la Figura **14.9.** Un controlador principal, denominado **gestor de monitorización de sensores,** reside en la parte superior de la estructura del programa y sirve para coordinar las siguientes funciones de control subordinadas:

- un controlador de procesamiento de la información de entrada denominado controlador de la entrada del sensor, coordina la recepción de todos los datos de entrada;
- un controlador del flujo de transformación, denominado controlador de las condiciones de alarma, supervisa todas las operaciones sobre los datos en su forma interna (por ejemplo; un módulo **que** invoca varios procedimientos de transformación de datos);

Información de configuración
Datos de configuración
Dar formato a identificación, tipo y localización
Generar visualización
Información del sensor
Leer sensores
Estado del sensor
Ajuste identificativo del sensor
Adquirir información de respuesta
Identificación del tipo de sensor, localización
Dar formato de visualización
Generar señal de alarma
Tipo de alarma
Establecer las condiciones de la alarma
Datos de alarma
Código de la condición de la alarma, identificación del sensor, información de temporización
Lista de números
Seleccionar el número del teléfono
Número de telefono
Establecer la conexión a la red telefónica
Tonos del número de telefono
Tono de línea número de telefono
Generar pulsos en la línea

FIGURA 14.8. Nivel 3 del DFD de *monitorizar sensores* con los límites de flujo.

- un controlador de procesamiento de información de salida, denominado controlador de la salida de alarma, coordina la producción de la información de salida.



No seas dogmático en esta parte. Podría ser necesario establecer dos o más controladores de procesamiento de entrada o computación, a causa de la complejidad del sistema a construir. Si el sentido común así te lo dicta, hazlo.

Aunque la Figura 14.9 implica una estructura con tres ramas, en grandes sistemas la complejidad del flujo puede hacer que existan dos o más módulos de control, uno para cada una de las funciones genéricas descritas anteriormente. El número de módulos del primer nivel debería limitarse al mínimo que pueda realizar las funciones de control y mantener al mismo tiempo unas buenas características de acoplamiento y cohesión.

Paso 6. Realizar «descomposición de segundo nivel». La descomposición de segundo nivel se realiza mediante la conversión de las transformaciones individuales (burbujas) de un DFD en los módulos correspondientes dentro de la arquitectura. Empezando desde el límite del centro de transformación y moviéndonos

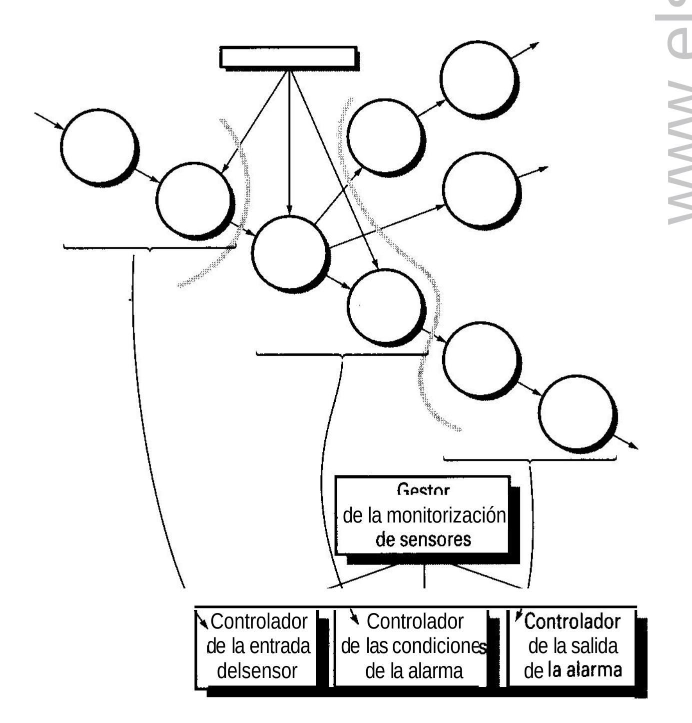


FIGURA 14.9. Descomposición de primer nivel para la monitorización de sensores.

hacia fuera a lo largo de los caminos de entrada, y luego de salida, las transformaciones se convierten en niveles subordinados de la estructura del software. El enfoque general del segundo nivel de descomposición del flujo de datos *HogarSeguro* se ilustra en la Figura **14.10.**

Aunque la Figura 14.10 ilustra una correspondencia una a uno entre las transformaciones del DFD y los módulos del software, ocurren frecuentemente diferentes conversiones. Se pueden combinar dos o incluso tres burbujas y representarlas como un solo módulo (teniendo presente los problemas potenciales de la cohesión), o una sola burbuja puede dividirse en dos o más módulos. Las consideraciones prácticas y las medidas de la calidad dictan el resultado de la descomposición en factores de segundo nivel. La revisión y el refinamiento pueden llevar a cambios en la estructura, pero puede servir como una primera iteración del diseño.



*Mantén bajos los **controladores** de **trabajo** en la estructura del programa. Así, la arquitectura será más fácil de **modificar**.*

La descomposición de factores de segundo nivel del flujo de entrada sigue de igual manera. La descomposición en factores se realiza de nuevo moviéndose hacia fuera desde el límite del centro de transformación correspondiente al flujo de entrada. El centro de transformación del software del subsistema de monitorizar sensores se direcciona de manera algo diferente. Cada

FIGURA 14.10. **Descomposición de factores de segundo nivel de monitorización de sensores.**

conversión de datos o transformaciones de cálculo de la porción de transformación del DFD se convierte en un módulo subordinado al controlador de transformación. En la Figura 14.11 se muestra una estructura de programa completa de primera iteración.

Los módulos direccionados de la manera descrita anteriormente y mostrados en la Figura 14.11 representan un diseño inicial de la estructura del programa. Aunque los módulos se nombran de manera que indiquen su función, se debería escribir para cada uno un breve texto que explique su procesamiento (adaptado del EP creado durante la modelación de análisis). El texto describe:

- la información que entra y la que sale fuera del módulo (una descripción de la interfaz);
- la información que es retenida en el módulo, por ejemplo, datos almacenados en una estructura de datos local;
- una descripción procedimental que indique los principales puntos de decisión y las tareas;
- un pequeño estudio de las restricciones y características especiales (por ejemplo, archivo I/O, características dependientes del hardware, requisitos especiales de tiempo).

El texto explicativo sirve como una primera generación de la especificación de diseño. Sin embargo, se dan más refinamientos y adiciones regularmente durante este periodo de diseño.

Paso 7. Refinar la estructura inicial de la arquitectura usando heurísticas para mejorar la calidad del software. Una primera estructura de arquitectura siempre puede refinarse aplicando los conceptos de independencia de módulos (Capítulo 13). **Los** módulos se incrementan o reducen para producir una descomposición razonable, buena cohesión, acoplamiento mínimo **y** lo más importante, una estructura que se pueda implementar sin dificultad, probarse sin confusión y mantenerse **sin** problemas.

Los refinamientos se rigen por el análisis y los métodos de evaluación descritos brevemente en la Sección 14.4, así como por consideraciones prácticas **y** por el sentido común. Hay veces, por ejemplo, que el controlador del flujo de datos de entrada es totalmente innecesario, o se requiere cierto procesamiento de entrada en un módulo subordinado al controlador de transformación, o no se puede evitar un alto acoplamiento debido a datos generales, o no se pueden lograr las características estructurales óptimas (vea la Sección 13.6). **Los** requisitos del software junto con el buen juicio son los árbitros finales.

CONSEJO

Elimina los módulos de control redundantes. Es decir, si un módulo de control no hace otro coso que controlar otro módulo, esto función de control debería explotarse a mayor nivel.

CONSEJO

Céntrese en lo independencia funcional de los módulos que derive. Su objetivo debe ser uno cohesión alta y un acoplamiento débil.

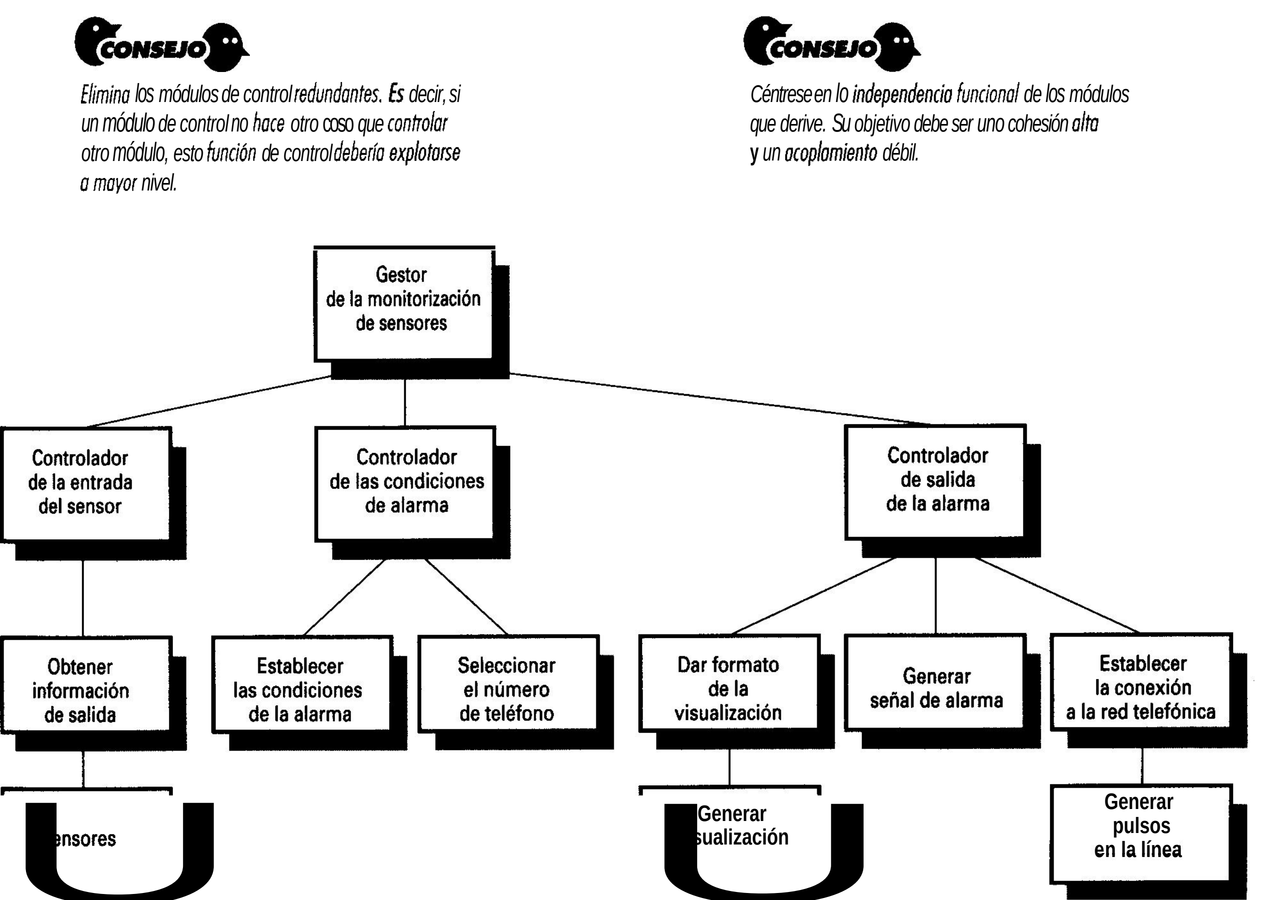


FIGURA 14.11. Primera iteración de la estructura del programa para *monitorizar* sensores.

Se pueden hacer muchas modificaciones a la primera estructura desarrollada para el subsistema de *monitorizar sensores* de *HogarSeguro.* Algunas de ellas son:

1. El controlador del flujo de entrada se puede eliminar porque es innecesario cuando sólo hay que manejar un solo camino de flujo de entrada.
2. La subestructura generada del flujo de transformación puede reducirse en el módulo **establecer las condiciones de alarma** (que incluirá ahora el procesamiento implicado por **seleccionar número de teléfono).** El controlador de transformación no será necesario y la pequeña disminución en la cohesión es tolerable.
3. Los módulos **dar formato de visualización** y **generar la visualización** pueden unirse (asumimos que dar formato de visualización es bastante simple) en un nuevo módulo denominado **producir visualización.**

En la Figura 14.12 se muestra la estructura del software refinada del subsistema de monitorizar sensores.

El objetivo de los siete puntos anteriores es desarrollar una representación general del software. Es decir, una vez que se ha definido la estructura, podemos evaluar y refinar la arquitectura del software viéndola en su conjunto. Las modificaciones hechas ahora requieren poco trabajo adicional, pero pueden tener un gran impacto en la calidad del software.

El lector debería reflexionar un momento y considerar la diferencia entre el enfoque de diseño descrito anteriormente y el proceso de «escribir programas)). Si el código es la única representación del software, el desarrollador tendrá grandes dificultades en evaluar o refinar a nivel general u holístico y, de hecho, tendrá dificultad para que «los árboles dejen ver el bosque».

14.7 ANÁLISIS DE LAS TRANSACCIONES

En muchas aplicaciones software, un solo elemento de datos inicia uno o varios de los flujos de información que llevan a cabo una función derivada por el elemento de datos iniciador. El elemento de datos, denominado transacción, y sus características de flujo correspondientes se trataron en la Sección 14.5.2. En esta sección consideramos los pasos de diseño usados para tratar el flujo de transacción.

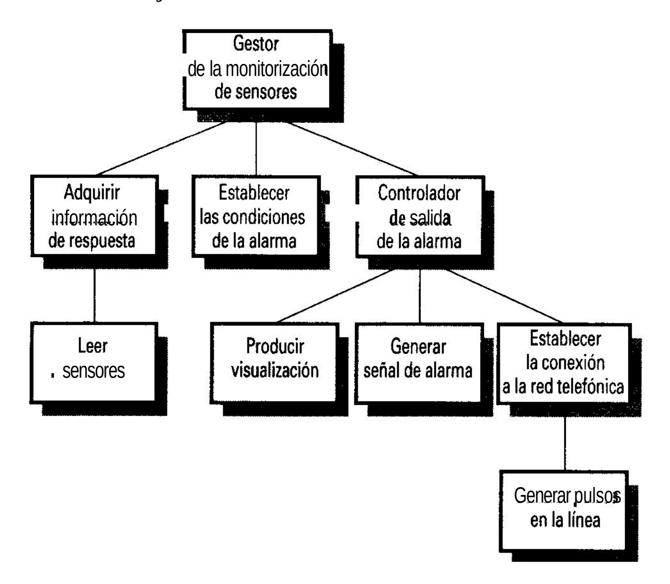


FIGURA 14.12. Estructura refinada del programa para *monitorizar sensores.*

14.7.1. Un ejemplo

El análisis de las transacciones se ilustrará considerando el subsistema de *interacción con el usuario* del software *HogarSeguro.* El nivel 1 del flujo de datos de este subsistema se muestra como parte de la Figura 14.6. Refinando el flujo, se desarrolla un nivel 2 del diagrama de flujo (también se crearían un diccionario de datos correspondiente, EC y EP) que se muestra en la Figura 14.13.

Como se muestra en la figura, **Órdenes y datos del usuario** fluye al sistema y provoca un flujo de información adicional a lo largo de uno de tres caminos de acción. Un elemento de datos, **tipo de orden,** hace que el flujo de datos se expanda del centro. Por tanto, la característica general del flujo de datos es de tipo transacción.

Debería recalcarse que el flujo de información a lo largo de dos de los tres caminos de acción acomodan flujos de entrada adicionales (por ejemplo, parámetros y datos del sistema son entradas en el camino de acción a «configurar»). Todos los caminos de acción fluyen a una sola transformación, *mostrar mensajes* y *estado.*

14.7.2. Pasos del diseño

Los pasos del diseño para el análisis de las transacciones son similares y en algunos casos idénticos a los pasos para el análisis de las transformaciones (Sección 14.6). La principal diferencia estriba en la conversión del DFD en la estructura del software.

Paso 1. Revisar el modelo fundamental del sistema.

Paso 2. Revisar y refinar los diagramas de flujo de datos para el software.

Paso 3. Determinar si el DFD tiene características de flujo de transformación o de transacción. Los pasos 1, 2 y 3 son idénticos a los correspondientes pasos del análisis de las transformaciones. El DFD mostrado en la Figura 14.13 tiene la clásica característica de flujo de tran-

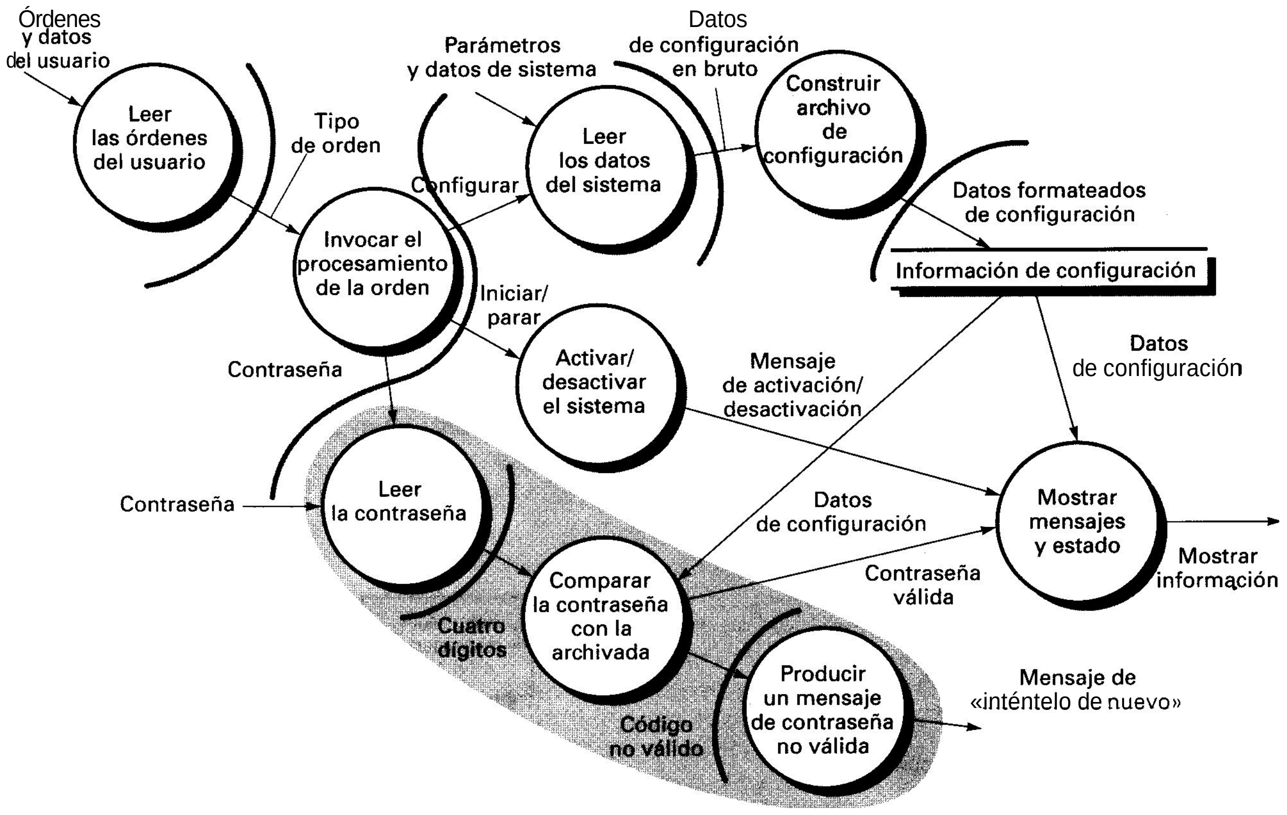


FIGURA 14.13. Nivel 2 de DFD para el subsistema de interación del usuario con límites del flujo.

sacción. Sin embargo, el flujo a lo largo de dos de los caminos de acción que emanan desde la burbuja *invocar el procesamiento de la orden* parece tener características de flujo de transformación. Por tanto, se deben establecer los límites de flujo para ambos tipos de flujos.

Paso 4. Identificar el centro de transacción y las características de flujo a lo largo de cada camino de acción. La posición transacción se puede discernir inmediatamente del DFD. El centro de transacción está en el origen de varios caminos de acción que fluyen radialmente desde él. Para el flujo mostrado en la Figura 14.13, la burbuja *invocar el procesamiento de la orden* es el centro de transacción.

El camino de entrada (por ejemplo el camino de flujo a lo largo del que se recibe una transacción) y todos los caminos de acción deben aislarse. Los límites que definen un camino de recepción y los caminos de acción también se muestran en la figura. Se debe evaluar las características individuales de flujo de cada camino de acción. Por ejemplo, el camino «de la contraseña» (mostrado incluido en un área sombreada en la Fig. 14.13) tiene características de transformación. Los flujos de entrada, de transformación y de salida se indican con límites.

Paso 5. Transformar el DFD en una estructura de programa adecuada al procesamiento de la transacción. El flujo de transacción se convierte en una arquitectura que contiene una rama de entrada y una rama de distribución. La estructura de la rama de entrada se desarrolla de la misma manera que para un análisis de transformación. Empezando por el centro de transacción, las burbujas que hay a lo largo del camino de entrada se convierten en módulos. La estructura de la rama de distribución contiene un módulo distribuidor que controla todos los módulos de acción subordinados. Cada camino de flujo de acción del DFD se convierte en una estructura que corresponde a sus características específicas de flujo. Este proceso se ilustra esquemáticamente en la Figura 14.14.



La descomposición de primer nivel tiene como resultado la derivación de la jerarquía de control para el software. La descomposición de segundo nivel distribuye los módulos de trabajo a los controladores apropiados.

Considerando el flujo de datos del subsistema de *interacción del usuario*, la descomposición de primer nivel del paso 5 se muestra en la Figura 14.15. Las burbujas *leer órdenes del usuario* y *activarldesactivar el sistema* se convierten directamente en la arquitectura, sin la necesidad de módulos intermedios de control. El centro de transacción, *invocar el procesamiento de la orden*, se convierte directamente en un módulo distribuidor con el mismo nombre. **Los** controladores de la configuración del sistema y procesamiento de la contraseña se obtienen como se indica en la Figura 14.16.

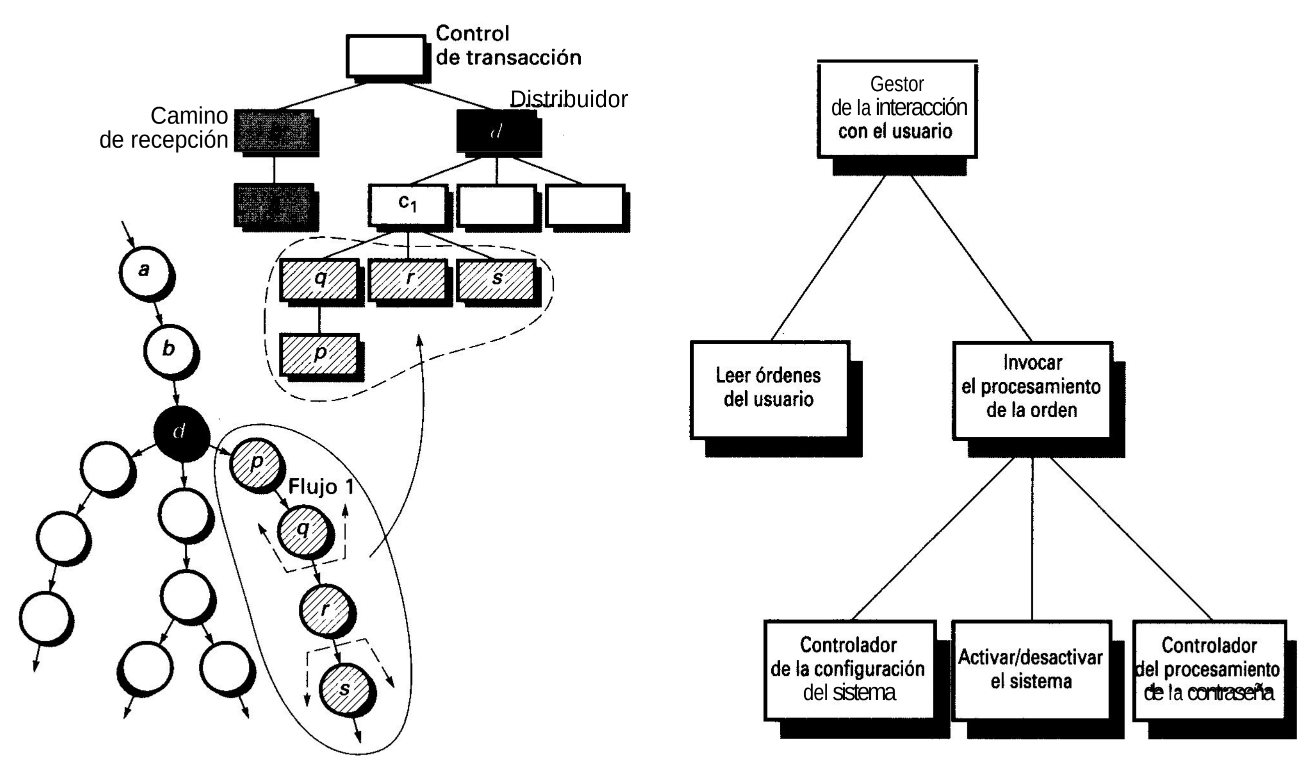


FIGURA 14.14. Análisis de transacción.

FIGURA 14.15. Descomposición en factores de primer nivel del subsistema interacción del usuario.

Gestor
de la interacción
del usuario

Leer órdenes
del usuario

Invocar
el procesamiento
de la orden

Controlador
de la configuración
del sistema

Activar/desactivar
el sistema

Controlador
del procesamiento
de la contraseña

Leer
datos
del sistema

Construir archivos
de
configuración

Leer la
contraseña

Comparar
la contraseña
con la
archivada

Controlador
de salida
de la contraseña

Mostrar
mensaje
y estado

Producir
un mensaje
de contraseña
no válida

FIGURA 14.16. Primera iteración de la estructura del programa del subsictema ***interacción del*** usuario.

www.elsolucionario.net

Paso 6. Descomponer y refinar la estructura de transacción y la estructura de todos los caminos de acción. Cada camino de acción del diagrama de flujo de datos tiene sus propias características de flujo de información. Ya hemos dicho anteriormente que se pueden encontrar flujos de transformación o de transacción. La «subestructura» relacionada con el camino de acción se desarrolla usando los pasos estudiados en esta sección y en la anterior.

Por ejemplo, considere el flujo de información del procesamiento de contraseña mostrado (dentro del área sombreada) en la Figura **14.13.** El flujo presenta las características clásicas de transformación. Se introduce una contraseña (flujo de entrada) y se transmite a un centro de transformación donde se compara con las contraseñas almacenadas. Se produce una alarma y un mensaje de advertencia (flujo de salida) si no coincide con ninguna.

El camino «configurar» se dibuja similarmente usando el análisis de transformación. La arquitectura de software resultante se muestra en la Figura 14.16.

Paso 7. Refinar la primera arquitectura del programa usando heurísticas de diseño para mejorar la calidad del software. Este paso del análisis de transacción es idéntico al correspondiente paso del análisis de transformación.

En ambos métodos de diseño se deben considerar cuidadosamente criterios tales como independencia del módulo, conveniencia (eficacia de implementación y prueba) y facilidad de mantenimiento a medida que se proponen modificaciones estructurales.

14.8 REFINAMIENTO DEL DISEÑO ARQUITECTÓNICO

El éxito de la aplicación del análisis de transformación o de transacción se complementa añadiendo la documentación adicional requerida como parte del diseño arquitectónico. Después de haber desarrollado y refinado la estructura, se deben completar las siguientes tareas:

- se debe desarrollar una descripción del procesamiento para cada módulo.
- se aporta una descripción de la interfaz para cada módulo.
- se definen las estructuras de datos generales y locales.
- se anotan todas las limitaciones/restricciones del diseño.
- se lleva a cabo una revisión del diseño.
- se considera un «refinamiento» (si es necesario y está justificado).

¿Qué pasa una vez que la arquitectura ha sido creada?

Un texto explicativo del procesamiento es (idealmente) una delimitada descripción sin ambigüedades del procesamiento que ocurre dentro de un módulo. La narrativa describe el procesamiento, las tareas, las decisiones y la entrada/salida. La descripción de la interfaz requiere el diseño de interfaces internas de módulo, interfaces externas del sistema y la interfaz hombre-computadora (Capítulo 15). El diseño de estructuras de datos puede tener un profundo impacto en la arquitectura y en los detalles procedimentales de cada componente del software. También se documentan las restricciones/limitaciones de cada módulo. Algunos aspectos típicos que pueden tratarse incluyen: la restricción de tipo o formato de datos, las limitaciones de memoria o de tiempo, la delimitación de los valores o cantidades de las estructuras de datos, los casos especiales no considerados y las características específicas de un módulo individual. El propósito de una sección restricciones/limitaciones es reducir el número de errores debidos a características funcionales asumidas.

Una vez que se ha desarrollado la documentación de diseño para todos los módulos, se lleva a cabo una revisión (vea las directrices de revisión en el Capítulo **8**). La revisión hace hincapié en el seguimiento de los requisitos del software, la calidad de la arquitectura del programa, las descripciones de las interfaces, las descripciones de las estructuras de datos, los datos prácticos de la implementación, la capacidad de prueba y la facilidad de mantenimiento.



Documento del diseño de software.

Debe fomentarse el refinamiento de la arquitectura del software durante las primeras etapas del diseño. Como ya dijimos anteriormente en este capítulo, los estilos arquitectónicos alternativos deben ser derivados, refinados y evaluados para un «mejor» enfoque. Este enfoque de optimización es uno de los verdaderos beneficios derivados del desarrollo de la representación de la arquitectura del software.

Es importante anotar que la simplicidad estructural es, a menudo, reflejo de elegancia y eficiencia. El refinamiento del diseño debería luchar por obtener un pequeño número de módulos consecuentes a la modularidad operativa y la estructura de datos menos compleja que sirva adecuadamente a los requisitos de información.

RESUMEN

La arquitectura del software nos proporciona una visión global del sistema a construir. Describe la estructura y la organización de los componentes del software, sus propiedades y las conexiones entre ellos. Los componentes del software incluyen módulos de programas y varias representaciones de datos que son manipulados por el programa. Además, el diseño de datos es una parte integral para la derivación de la arquitectura del software. La arquitectura marca decisiones de diseño tempranas y proporciona el mecanismo para evaluar los beneficios de las estructuras de sistema alternativas.

El diseño de datos traduce los objetos de datos definidos en el modelo de análisis, en estructuras de datos que residen dentro del software. Los atributos que describe el objeto, las relaciones entre los objetos de datos y su uso dentro del programa influyen en la elección de la estructura de datos. A mayor nivel de abstracción, el diseño de datos conducirá a lo que se define como una arquitectura para una base de datos o un almacén de datos.

El ingeniero del software cuenta con diferentes estilos y patrones arquitectónicos. Cada estilo describe una categoría de sistema que abarca un conjunto de componentes que realizan una función requerida por el sistema, un conjunto de conectores que posibilitan la comunicación, la coordinación y cooperación entre los componentes, las restricciones que definen cómo se integran los componentes para conformar el sistema, y los modelos semánticos que facilitan al diseñador el entendimiento de todas las partes del sistema.

Han sido propuestos uno o varios estilos arquitectónicos por sistema, y el método de análisis de compromisos para la arquitectura podría utilizarse para evaluar la eficacia de cada arquitectura propuesta. Esto se consigue determinando la sensibilidad de los atributos de calidad seleccionados (también llamados dimensiones del diseño) con diferentes mecanismos de realización que reflejan las propiedades de la arquitectura.

El método de diseño arquitectónico presentado en este capítulo utiliza las características del flujo de datos descritas en el modelo de análisis que derivan de un estilo arquitectónico utilizado comúnmente. El diagrama de flujo de datos se descompone dentro de la estructura del sistema a través de dos enfoques de análisis—el análisis de las transformaciones y/o el análisis de las transacciones—. Se aplica el análisis de las transformaciones a un flujo de información que presenta diferentes límites entre los datos de entrada y de salida. El DFD se organiza en una estructura que asigna los controles de entrada, procesamiento y salida a través de tres jerarquías separadas de módulos de descomposición en factores. El análisis de las transformaciones se aplica cuando un Único ítem de información bifurca su flujo a través de diferentes caminos. El DFD se organiza en una estructura que asigna el control a una subestructura que adquiere y evalúa la transacción. Otra subestructura controla todas las acciones potenciales de procesamiento basadas en la transacción. Una vez que la arquitectura ha sido perfilada se elabora y se analiza contrastándola con los criterios de calidad.

El diseño arquitectónico agrupa un grupo inicial de actividades de diseño que conducen a un modelo completo del diseño del software. En los siguientes capítulos, se estudiará el diseño de las interfaces y de los componentes.

REFERENCIAS

[AHO83] Aho, A.V., J. Hopcroft y J.Ullmann, *Data Structures and Algorithms,* Addison-Wesley, 1983.

[BAS98] Bass, L., P. Clements y R. Kazman, *Software Architecture in Practice,* Addison-Wesley, 1998.

[DAH72] Dah, O., E. Dijkstra y C. Hoare, *Structured Programming,* Academic Press, 1972.

[DAT95] Date, C.J., *An Introduction to Database Systems,* Sexta Edición, Addison-Wesley, 1995.

[DEN73] Dennis, J.B., «Modularity», en *Advanced Course on Software Engineering,* F.L. Bauer (ed.), Springer-Verlag, Nueva York, 1973, pp. 128-182.

[FRE80] Freeman, P., «The Context of Design», in *Software Design Techniques* (L.P. Freeman y A. Wasserman, eds.), IEEE Computer Society Press, 3."ed., 1980, pp. 2-4.

[INM95] Inmon, W.H., «What is a Data Warehouse?» Prism Solutions, Inc. 1995, presentada en: **http://www.cait.wustl.edu/cait/papers/prism/vol1_no1**.

[KAZ98] Kazman, R. *The Architectural Tradeoff Analysis Method,* Software Engineering Institute, CMU/SEI-98-TR-008. Julio 1998.

[KIM98] Kimball, R., L. Reeves, M. Ross y W. Thornthwaite, *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying,* Data Warehouses, Willey, 1998.

[LIN79] Linger, R.C., H.D. Mills y B.I. Witt, *Structured Programming,* Addison-Wesley, 1979.

[MAT96] Mattison, R., *Data Warehousing: Strategies, Technologies and Techniques,* McGraw-Hill, 1996.

[MOR80] Morris, J., «Programming by Successive Refinement of Data Abstractions», *Software-Practice and Experience,* vol. 10, núm. 4, abril 1980, pp. 249-263.

[MYE78] Myers, G., *Composite Structures Design,* Van-Nostrand, 1978.

[PET81] Peters, L.J., ***Software Design: Methods and Techniques,*** Yourdon Press, Nueva York, 1981.

[PRE98] Preiss, B.R. ***Data Structures and Algorithms: With Ohject-Oriented Design Patterns in C++***, Wiley, 1998.

[SHA96] Shaw, M., y D. Garlan, ***Software Arquitecture,*** Prentice Hall, 1996.

[SHA97] Shaw, M., y P. Clements, «A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems», ***Proc. COMPSAC,*** Washington DC, Agosto 1997.

[STE74] Stevens, W., G. Myers y L. Constantine, «Structured Design», ***IBM System Journal,*** vol.13, n.° 2, 1974, pp. 115-139.

[WAS80] Wasserman, A., «Principles of Systematic Data Design and Implementation», en ***Software Design Techniques*** (P. Freeman y A. Wasserman, eds.), 3.ª ed., IEEE Computer Society Press, 1980, pp. 287-293.

[WIR71] Wirth, N., «Program Development by Stepwise Refinement», CACM, vol. 14, n.° 4, 1971, pp. 221-227.

[YOU79] Yourdon, E., y L. Constantine, ***Structures Design,*** Prentice-Hall, 1979.

[ZHA98] Zhao, J., «On Assessing the Complexity of Software Architectures», ***Proc. Intl. Software Architecture Workshop***, ACM, Orlando, Florida, 1998, pp. 163-167.

PROBLEMAS Y PUNTOS A CONSIDERAR

14.1. Usando la arquitectura de una casa o un edificio a modo de metáfora, dibuje comparaciones con la arquitectura del software. ¿En qué se parecen la disciplina de la arquitectura clásica y la de la arquitectura del software? ¿En qué se diferencian?

14.2. Escriba un documento de tres a cinco páginas que contenga directrices para la selección de estructuras de datos basándose en la naturaleza del problema. Empiece delimitando las clásicas estructuras de datos que se encuentran en el software y después describiendo los criterios para la selección de éstos para tipos particulares de problemas.

14.3. Explique la diferencia entre una base de datos que sirve a una o más aplicaciones de negocios convencionales y un almacén de datos.

14.4. Escriba un documento de tres a cinco página que describa cómo se utilizan las técnicas de minería de datos en un entorno de negocio y el estado actual de las técnicas DCBC.

14.5. Presente dos o tres ejemplos de aplicaciones para cada estilo arquitectónico citado en la Sección 14.3.1.

14.6. Algunos de los estilos arquitectónicos citados en la Sección 14.3.1. son jerárquicos por naturaleza y otros no. Haga una lista de cada tipo. ¿Cómo se implementan los estilos arquitectónicos no jerárquicos?

14.7. Seleccione una aplicación que le sea familiar. Conteste cada una de las preguntas propuestas para el control y los datos de la Sección 14.3.2.

14.8. Estudie el MACA (utilizando el libro de [KAZ98]) y presente un estudio detallado de los seis pasos presentados en la Sección 14.4.1.

14.9. Seleccione una aplicación que le sea familiar. Utilizando, donde sea requerido, su mejor intuición, identifique el conjunto de dimensiones del diseño y después realice el análisis del espectro y el análisis de la selección del diseño.

14.10. Estudie el EDC (utilizando el libro de [SHA96]) y desarrolle un espacio de diseño cuantificado para una aplicación que le sea familiar.

14.11. Algunos diseñadores defienden que todo el flujo de datos debe ser tratado como orientado a transformaciones. Estudie como afectará esta opinión a la estructura del software que se obtiene cuando un flujo orientado a transacción es tratado como de transformación. Utilice un flujo de ejemplo para ilustrar los puntos importantes.

14.12. Si no lo ha hecho, complete el Problema 12.12. Utilice los métodos de diseño descritos en este capítulo para desarrollar una estructura de programa para el SSRB.

14.13. Mediante un diagrama de flujo de datos y una descripción del procesamiento, describa un sistema basado en computadora que tenga unas características de flujo de transformación singulares. Defina los límites del flujo y transforme el DFD en la estructura software usando una técnica de las descritas en la Sección 14.6.

14.14. Mediante un diagrama de flujo de datos y una descripción de procesamiento, describa un sistema basado en computadora que tenga unas características de flujo de transacción claras. Defina los límites del flujo y direcciones el DFD en una estructura software utilizando la técnica descrita en la Sección 14.7.

14.15. Usando los requisitos obtenidos en un estudio hecho en clase, complete los DFD y el diseño arquitectónico para el ejemplo *HogarSeguro* presentado en las Secciones 14.6 y 14.7. Valore la independencia funcional de todos los módulos. Documente su diseño.

14.16. Estudie las ventajas y dificultades relativas de aplicar un diseño orientado al flujo de datos en las siguientes áreas: (a) aplicaciones de microprocesador empotrado, (b) análisis de ingeniería/científico, (c) gráficos por computadora, (d) diseño de sistemas operativos, (e) aplicaciones de negocio, (f) diseño de sistemas de gestión de bases de datos, (g) diseño de software de comunicaciones, (h) diseño de compiladores, (i) aplicaciones de control de proceso y (j) aplicaciones de inteligencia artificial.

14.17. Dado un conjunto de requisitos que le proporcione su profesor (o un conjunto de requisitos de un problema en el que esté trabajando actualmente) desarrolle un diseño arquitectónico completo. Lleve a cabo una revisión del diseño (Capítulo 8) para valorar la calidad de su diseño. Este problema debe asignarse a un equipo en vez de a un solo individuo.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Durante la Última década han aparecido muchísimos libros sobre arquitectura de software. Los libros de Shaw y Gannon [SHA96], Bass, Clements y Kazman [BAS98] y Buschmann y colaboradores[BUS98], proporcionan un tratamiento en profundidad sobre la materia. El primer trabajo de Garlan ***(An Introduction to Software Architecture,*** Software Engineering Institute, CMU/SEI-94-TR-02 1,1994) proporciona una excelente introducción al tema.

Los libros específicos de implementación de arquitectura, sitúan el diseño arquitectónico dentro de un entorno específico de desarrollo ***o*** tecnología. Mowray ***(CORBA Design Patterns,*** Wiley, 1997) y Mark y colaboradores ***(Object Management Architecture Guide,*** Wiley, 1996) proporcionan parámetros de diseño detallados para el marco de soporte de las aplicaciones distribuidas de CORBA. Shanley ***(Protected Mode Software Architecture,*** Addison-Wesley, 1996) proporciona una guía de diseño arquitectónico para aquellos que diseñen sistemas operativos a tiempo real basados en computadora, sistemas operativos multitarea ***o*** controladores de dispositivos.

Las investigaciones actuales sobre arquitectura de software están recogidas en el anuario ***Proceedings of the International Workshop on Software Architecture,*** patrocinado por la ACM y otras organizaciones de computadoras y en el ***Proceedings of the International Conference on Software Engineering.***

El modelado de datos es un prerrequisito para un buen diseño de datos. **Los** libros de Teory ***(Database Modeling & Design,*** Academic Press, 1998), Schimdt ***(Data Modeling for Information Professionals,*** Prentice Hall, 1998), Bobak, ***(Data Modeling and Design for Today's Architectures,*** Artech House, 1997), Silverston, Graziano e Inmon ***(The Data Model Resource Book,*** Wiley, 1997), Date [DAT95], Reingruber y Gregory ***(The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models,*** Wiley, 1994), Hay ***(Data Model Patterns: Conventions of Thought,*** Dorset House, 1994) contienen una presentación detallada de la notación de modelado de datos, heurísticas, y enfoques de diseño de bases de datos. En los últimos años el diseño de almacenes de datos ha ido cobrando importancia. Los libros de Humphreys, Hawkins y Dy ***(Data Warehousing: Architecture and Implementation,*** Prentice Hall, 1999), Kimball [KIM98] e Inmon [INM95] cubren con gran detalle la materia.

Docenas de libros actuales, a menudo abordan el diseño de datos y el diseño de estructuras desde un contexto específico del lenguaje de programación. Algunos ejemplos típicos los encontramos en:

Horowitz, **E.** Y S. Sahni, ***Fundamentals of Data Structures in Pascal,*** 4.ª ed., W.H. Freeman & Co., 1999.

Kingston, J.H., ***Algorithms and Data Structures: Design, Correctness, Analysis,*** 2.ª ed., Addison-Wesley, 1997.

Main, **M.,** ***Data Structures & Other Objects Using Java,*** Addison-Wesley, 1998.

Preiss, B.R., ***Data Structures and Algorithms: With Object-Oriented Design Patterns in C++,*** Wiley, 1998.

Sedgewick, **R.,** ***Algorithms in C++: Fundamentals, Data Structures, Sorting, Searching,*** Addison-Wesley, 1999.

Standish, T.A., ***Data Structures in Java,*** Addison-Wesley, 1997.

Standish, T.A., ***Data Structures, Algorithms, and Software Principles in C,*** Addison-Wesley, 1995.

En la mayoría de **los** libros dedicados a la ingeniería de software se puede encontrar un tratamiento general del diseño del software en discusión con el diseño arquitectónico y el diseño de datos. **Los** libros de Pfleeger ***(Software Engineering: Theory and Practice,*** Prentice may, 1998) y Sommerville (***Software Engineering,*** 5." ed., Addison-Wesley, 1995) son representativos de los libros que cubren en detalle el tema del diseño.

Se puede encontrar un tratamiento más riguroso de la materia en Feijs ***(Formalization of Design Methods,*** Prentice may, 1993), Witt y colaboradores ***(Software Architecture and Design Principles,*** Thomson Publishing, 1994) y Budgen ***(Software Design,*** Addison-Wesley, 1994).

Se pueden encontrar representaciones completas de diseños orientados a flujos de datos en Myers [MYE78], Yourdon y Constantine [YOU79], Buhr ***(System Design with Ada,*** Prentice may, 1984), y Page-Jones ***(The Practica1 Guide to Structured Systems Design,*** segunda edición, Prentice may, 1988). Estos libros están dedicados solamente al diseño y proporcionan unas completas tutorías en el enfoque de flujo de datos.

En Intemet están disponibles una gran variedad de fuentes de información sobre diseño de software y temas relacionados. Una lista actualizada de referencias web sobre conceptos y métodos de diseño relevantes se puede encontrar en: **http://www.pressman5.com**.

CAPÍTULO 15 DISEÑO DE LA INTERFAZ DE USUARIO

EL plano de una casa (su diseño arquitectónico) no está completo sin la representación de puertas, ventanas y conexiones de servicios para el agua, electricidad y teléfono (por no mencionar la televisión por cable). Las «puertas, ventanas y conexiones de servicios» del software informático es lo que constituye el diseño de la interfaz de usuario.

El diseño de la interfaz se centra en tres áreas de interés: (1) el diseño de la interfaz entre los componentes del software; (2) el diseño de las interfaces entre el software y los otros productores y consumidores de información no humanos (esto es, otras entidades externas) y (3) el diseño de la interfaz entre el hombre (esto es, el usuario) y la computadora. En este capítulo nos centraremos exclusivamente en la tercera categoría de diseño de interfaz —el diseño de la interfaz de usuario—.

Ben Shneiderman [SHN87] habla sobre esta categoría de diseño en el prólogo de su libro y afirma lo siguiente:

> **Para muchos usuarios de sistemas de información computerizados la frustración y la ansiedad forman parte de su vida diaria. Luchan por aprender el lenguaje de órdenes y los sistemas de selección de menús que supuestamente les ayudan a realizar su trabajo. Algunas personas se encuentran con casos tan serios de shocks informáticos, terror en el terminal o neurosis en la red, que evitan utilizar sistemas computerizados.**

VISTAZO RÁPIDO

¿Qué es? El diseño de la interfaz de usuario es la categoría de diseño que crea un medio de comunicación entre el hombre y la máquina. Con un conjunto de principios para el diseño de la interfaz, el diseño identifica los objetos y acciones de la interfaz y crea entonces un formato de pantalla que formará la base del prototipo de interfaz de usuario.

¿Quién lo hace? El ingeniero del software es quien diseña la interfaz de usuario mediante la aplicación del proceso iterativo que se sirve de los principios predefinidos del diseño.

¿Por qué es importante? Si el software es difícil de utilizar, si obliga a cometer errores, o causa frustración para conseguir los objetivos, no será de agrado, independientemente de la potencia informática que demuestre o de la funcionalidad que ofrezca. Dado que la interfaz es la que da forma a la percepción del software por parte del usuario, tiene que estar bien diseñada.

¿Cuáles son los pasos? El diseño de la interfaz de usuario comienza con la identificación de los requisitos del usuario, de la tarea y del entorno. Una vez identificadas las tareas, se crean y se analizan los escenarios del usuario para definir el conjunto de objetos y de acciones de la interfaz. Esto es lo que forma la base para la creación del formato de la pantalla que representa el diseño gráfico y la colocación de iconos, la definición del texto descriptivo en pantalla, la especificación y títulos de las ventanas, y la especificación de los elementos principales y secundarios del menú. Las herramientas se utilizan para generar prototipos y por último implementar el modelo de diseño y evaluar la calidad del resultado.

¿Cuál es el producto obtenido? La creación de escenarios de usuarios y la generación de formatos de pantallas. Y el desarrollo y modificación iterativo de prototipos.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Los usuarios controlan el prototipo mediante pruebas y la respuesta obtenida del control del texto se utiliza para la siguiente modificación iterativa del prototipo.

Los problemas a los que alude Shneiderman son reales. Es cierto que las interfaces gráficas, ventanas, iconos y selecciones mediante ratón han eliminado muchos de los terribles problemas con la interfaz. Pero incluso en un «mundo de ventanas» todos encontramos interfaces de usuario difíciles de aprender, difíciles de utilizar, confusas, imperdonables y en muchos casos totalmente frustrantes. Sin embargo, hay quien dedica tiempo y energías construyendo estas interfaces, y es posible que estos problemas no los crearan a propósito.

15.1 LAS REGLAS DE ORO

Theo Mantel [MAN97] en su libro crea tres «reglas de oro» para el diseño de la interfaz:

1. Dar el control al usuario
2. Reducir la carga de memoria del usuario
3. Construir una interfaz consecuente

Estas reglas de oro forman en realidad la base para los principios del diseño de la interfaz de usuario que servirán de guía para esta actividad de diseño de software tan importante.

15.1.1. Dar el control al usuario

Durante la sesión de recopilación de los requisitos para un nuevo sistema de información, un usuario clave fue preguntado a cerca de los atributos de la interfaz gráfica orientada a ventanas.

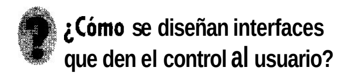
El usuario respondió solemnemente, «Lo que me gustaría realmente es un sistema que lea mi mente. Que conozca lo que quiero hacer antes de necesitarlo y que me facilite hacerlo. Eso es todo. Simplemente eso.»

Mi primera reacción fue mover la cabeza y sonreír, y hacer una pausa por unos instantes. No había nada malo en la solicitud del usuario. Lo que quería era que un sistema reaccionara ante sus necesidades y que le ayudara a hacer las cosas. Quería controlar la computadora, y no dejar que la computadora le controlara.

La mayor parte de las restricciones y limitaciones impuestas por el diseñador se han pensado para simplificar el modo de interacción. Pero, ¿para quienes? En muchos casos es posible que el diseñador introduzca limitaciones y restricciones para simplificar la implementación de la interfaz. Y el resultado puede ser una interfaz fácil de construir, pero frustrante de utilizar.

Mandel [MAN97] define una serie de principios de diseño que permiten dar control al usuario:

Definir los modos de interacción de manera que no obligue a que el usuario realice acciones innecesarias y no deseadas. Un modo de interacción es el estado actual de la interfaz. Por ejemplo, si en el procesador de textos se selecciona el *corrector ortográfico*, el software pasa a modo corrector ortográfico. No hay ninguna razón por la que obligar a que el usuario permanezca en este modo si el usuario desea continuar editando una parte pequeña de texto. El usuario deberá tener la posibilidad de entrar y salir de este modo sin mucho o ningún esfuerzo.



Tener en consideración una interacción flexible. Dado que diferentes usuarios tienen preferencias de interacción diferentes, se deberán proporcionar diferentes selecciones. Por ejemplo, un software que pueda permitir al usuario interactuar a través de las órdenes del teclado, con el movimiento del ratón, con un lápiz digitalizador, mediante órdenes para el reconocimiento de voz. Sin embargo, no toda acción responde a todo mecanismo de interacción. Considere por ejemplo, la dificultad de utilizar órdenes del teclado (o entrada de voz) para dibujar una forma compleja.

Permitir que la interacción del usuario se pueda interrumpir y deshacer. Cuando un usuario se ve involucrado en una secuencia de acciones, deberá poder interrumpir la secuencia para hacer cualquier otra cosa (sin perder el trabajo que se hubiera hecho anteriormente). El usuario deberá también tener la posibilidad de «deshacer» cualquier acción.

Aligerar la interacción a medida que avanza el nivel de conocimiento y permitir personalizar la interacción. El usuario a menudo se encuentra haciendo la misma secuencia de interacciones de manera repetida. Merece la pena señalar un mecanismo de «macros» que posibilite al usuario personalizar la interfaz y así facilitar la interacción.



Uno de los errores más comunes que se comete cuando intentamos diseñar algo a prueba de tontos es subestimar la ingenuidad de los tontos.
Douglas Adams

Ocultar al usuario ocasional los entresijos técnicos. La interfaz de usuario deberá introducir al usuario en el mundo virtual de la aplicación. El usuario no tiene que conocer el sistema operativo, las funciones de gestión de archivos, o cualquier otro secreto de la tecnología informática. Esencialmente, la interfaz no deberá requerir nunca que el usuario interactúe a un nivel «interno» de la máquina (por ejemplo, el usuario no tendrá que teclear nunca las órdenes del sistema operativo desde dentro del software de aplicación).

Diseñar la interacción directa con los objetos que aparecen en la pantalla. El usuario tiene un sentimiento de control cuando manipula los objetos necesarios para llevar a cabo una tarea de forma similar a lo que ocurriría si el objeto fuera algo físico. Como ejemplo de manipulación directa puede ser una interfaz de aplicación que permita al usuario «alargar» un objeto (cambiar su tamaño).

15.1.2. Reducir la carga de memoria del usuario

Cuanto más tenga que recordar un usuario, más propensa a errores será su interacción con el sistema. Esta es la razón por la que una interfaz de usuario bien diseñada no pondrá a prueba la memoria del usuario. Siempre que sea posible, el sistema deberá «recordar» la información per-

tinente y ayudar a que el usuario recuerde mediante un escenario de interacción. Mandel [MAN97] define los principios de diseño que hacen posible que una interfaz reduzca la carga de memoria del usuario:

Reducir la demanda de memoria a corto plazo. Cuando los usuarios se ven involucrados en tareas complejas, exigir una memoria a corto plazo puede ser significativo. La interfaz se deberá diseñar para reducir los requisitos y recordar acciones y resultados anteriores. Esto se puede llevar a cabo mediante claves visuales que posibiliten al usuario reconocer acciones anteriores sin tenerlas que recordar.

Establecer valorespor defecto útiles. El conjunto inicial de valores por defecto tendrá que ser de utilidad para al usuario, pero un usuario también deberá tener la capacidad de especificar sus propias preferencias. Sin embargo, deberá disponer de una opción de «reinicialización» que le permita volver a definir los valores por defecto.

Definir las deficiencias que sean intuitivas. Cuando para diseñar un sistema se utiliza la mnemónica (por ejemplo, alt-P para invocar la función de imprimir), ésta deberá ir unida a una acción que sea fácil de recordar (por ejemplo, la primera letra de la tarea que se invoca).

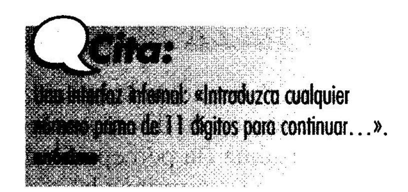
¿Cómo se pueden diseñar interfaces que reduzcan la carga de memoria del usuario?

El formato visual de la interfaz se deberá basar en una metáfora del mundo real. Por ejemplo, en un sistema de pago de facturas se deberá utilizar la metáfora de la chequera y el registro del cheque para conducir al usuario por el proceso del pago de facturas. Esto hace posible que el usuario comprenda bien las pistas y que no tenga que memorizar una secuencia secreta de interacciones.

Desglosar la información de forma progresiva. La interfaz deberá organizarse de forma jerárquica. Esto es, en cualquier información sobre una tarea se deberá presentar un objeto o algún comportamiento en primer lugar a un nivel alto de abstracción. Y solo después de que el usuario indique su preferencia realizando la selección mediante el ratón se presentarán más detalles. Un ejemplo muy común en muchas aplicaciones de procesamiento de texto es la función de subrayado dado que es una función que pertenece al menú de estilo de texto. Sin embargo no se muestran todas las posibilidades de subrayado. El usuario es el que debe seleccionar el subrayado, y así se presentarán entonces las opciones de esta función (por ejemplo, subrayado sencillo, subrayado doble, subrayado de guiones).

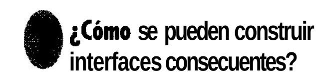
15.1.3. Construcción de una interfaz consistente

La interfaz deberá adquirir y presentar la información de forma consecuente. Esto implica (1) que toda la información visual esté organizada de acuerdo con el



diseño estándar que se mantiene en todas las presentaciones de pantallas; (2) que todos los mecanismos de entrada se limiten a un conjunto limitado y que se utilicen consecuentemente por toda la aplicación, y que (3) los mecanismos para ir de tarea a tarea se hayan definido e implementado consecuentemente. Mandel [MAN97] define un conjunto de principios de diseño que ayudar a construir una interfaz consecuente:

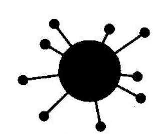
Permitir que el usuario realice una tarea en el contexto adecuado. Muchas interfaces implementan capas complejas de interacciones con docenas de imágenes de pantallas. Es importante proporcionar indicadores (por ejemplo, títulos de ventanas, iconos gráficos, codificaciones en colores consecuentes) que posibiliten al usuario conocer el contexto del trabajo que está llevando a cabo. Además, el usuario deberá ser capaz de determinar de dónde procede y qué alternativas existen para la transición a una tarea nueva.



Mantener la consistencia en toda la familia de aplicaciones. Un conjunto de aplicaciones (o productos) deberá implementar las mismas reglas de diseño para mantener la consistencia en toda la interacción.

Los modelos interactivos anteriores han esperanzado al usuario, no realicemos cambios a menos que exista alguna razón convincente para hacerlo. Una vez que una secuencia interactiva se ha convertido en un estándar hecho (por ejemplo, la utilización de alt-S para grabar un archivo), el usuario espera utilizar esta combinación en todas las aplicaciones que se encuentre. Un cambio podría originar confusión (por ejemplo, la utilización de alt-S para invocar la función cambiar de tamaño).

Los principios del diseño de interfaces tratados aquí y en sesiones anteriores proporcionan una guía básica para la ingeniería del software. En la siguiente sección examinaremos el proceso de diseño de la interfaz.



Líneas Generales poro el diseño de lo interíoces.

El proceso global para el diseño de la interfaz de usuario comienza con la creación de diferentes modelos de funcionamiento del sistema (la percepción desde fuera). Es entonces cuando se determinan las tareas orientadas al hombre y a la máquina que se requieren para lograr el funcionamiento del sistema; se tienen en consideración los temas de diseño que se aplican a todos los diseños de interfaces; se utilizan herramientas para generar prototipos y por último para implementar el modelo de diseño, y evaluar la calidad del resultado.

15.2.1. Modelos de diseño de la interfaz

Cuando se va a diseñar la interfaz de usuario entran en juego cuatro modelos diferentes. El ingeniero del software crea un *modelo de diseño;* cualquier *otro* ingeniero (o el mismo ingeniero del software) establece un *modelo de usuario,* el usuario final desarrolla una imagen mental que se suele llamar *modelo de usuario o percepción del usuario,* y los que implementan el sistema crean una *imagen de sistema* [RUBSS]. Desgraciadamente, todos y cada uno de los modelos pueden diferir significativamente. El papel del diseñador de interfaz es reconciliar estas diferencias y derivar una representación consecuente de la interfaz.

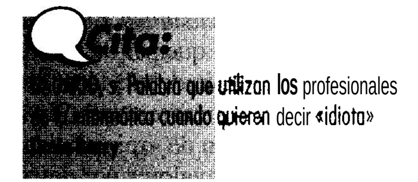


Una fuente excelente de directrices de diseño y referencias se puede encontrar en **www.ibm.com/ibm/easy/**

Un modelo de diseño de un sistema completo incorpora las representaciones del software en función de **los** datos, arquitectura, interfaz y procedimiento. La especificación de los requisitos puede que establezca ciertas limitaciones que ayudarán a definir al usuario del sistema, pero el diseño de la interfaz suele ser un único tema secundario de modelo de interfaz'.

El modelo de usuario representa el perfil de los usuarios finales del sistema. Para construir una interfaz de usuario efectiva, «todo diseño deberá comenzar por conocer los usuarios destino, así como los perfiles de edad, sexo, habilidades físicas, educación, antecedentes culturales o étnicos, motivación, objetivos y personalidad» [SCH87] Además de esto se pueden establecer las siguientes categorías de usuarios:

- **principiantes:** en general no tienen *conocimientos sintáctico?* ni *conocimientos semánticos*[3] de la utilización de la aplicación o del sistema;
- **usuarios esporádicos y con conocimientos:** poseen un conocimiento semántico razonable, pero una retención baja de la información necesaria para utilizar la interfaz;
- **usuarios frecuentes y con conocimientos:** poseen el conocimiento sintáctico y semántico suficiente como para llegar al «síndrome del usuario avanzado)), esto es, individuos que buscan interrupciones breves y modos abreviados de interacción.



La percepción del sistema (el modelo de usuario) es la imagen del sistema que el usuario final tiene en **su** mente. Por ejemplo, si se preguntara a un usuario de un procesador de texto en particular que describiera su **forma** de manejar el programa, la respuesta vendría guiada por la percepción del sistema. La precisión de la descripción dependerá del perfil del usuario (por ejemplo, los principiantes harían lo posible por responder con una respuesta muy elemental) y de la familiaridad global con el software del dominio de la aplicación. Un usuario que comprenda por completo los procesadores de texto, aunque puede que haya trabajado solo una vez con ese procesador específico, es posible que proporcione de verdad una descripción más completa de **su** funcionamiento que el principiante que haya pasado unas cuantas semanas intentando aprender el funcionamiento del sistema.

La imagen del sistema es una combinación de fachada externa del sistema basado en computadora (la apariencia del sistema) y la información de soporte (libros, manuales, cintas de vídeo, archivos de ayuda) todo lo cual ayuda a describir la sintaxis y la semántica del sistema. Cuando la imagen y la percepción del sistema coinciden, los usuarios generalmente se sienten a gusto con el software y con su funcionamiento. Para llevar a cabo esta «mezcla» de modelos, el modelo de diseño deberá desarrollarse con el fin de acoplar la información del modelo de usuario, y la imagen del sistema deberá reflejar de forma precisa la información sintáctica y semántica de la interfaz.

Los modelos descritos anteriormente en esta sección son «abstracciones de lo que el usuario está haciendo o piensa que está haciendo o de lo que cualquier otra per-

[1] Por supuesto esto no es como debena ser. Para sistemas interactivos, el diseño de la interfaz es tan importante como el diseño de los datos, arquitectura o el de componentes.

[2] En este contexto el conocimiento sintáctico se refiere a la mecánica de interacción que *se* requiere para utilizar la interfaz de forma eficaz.

[3] El conocimiento semántico se refiere al sentido subsiguiente de aplicación —una comprensión de la realización de todas las funciones, del significado de entrada y salida, de las metas y objetivos del sistema—.

sona piensa que debería estar haciendo cuando utiliza un sistema interactivo» [MON84]. Esencialmente, estos modelos permiten que el diseñador de la interfaz satisfaga un elemento clave del principio más importante del diseño de la interfaz de usuario: «quien conoce al usuario, conoce las tareas».



Cuando la imagen del sistema y la petcepcibn del sistema coinciden, el usuario puede utilizar la aplicacibn de forma efectiva.

15.2.2. El proceso de diseño de la interfaz de usuario

El proceso de diseño de las interfaces de usuario es iterativo y se puede representar mediante un modelo espiral similar al abordado en el Capítulo 2. En la Figura 15.1 se puede observar que el proceso de diseño de la interfaz de usuario acompaña cuatro actividades distintas de marco de trabajo [MAN97]:

1. Análisis y modelado de usuarios, tareas y entornos.
2. Diseño de la interfaz
3. Implementación de la interfaz
4. Validación de la interfaz

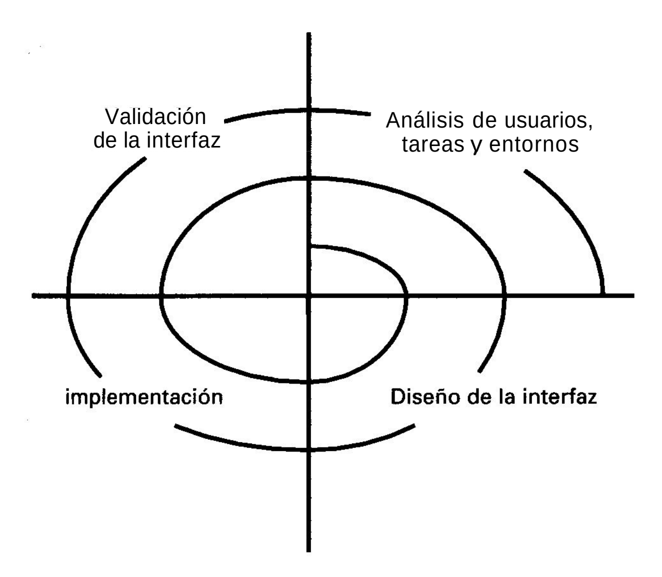
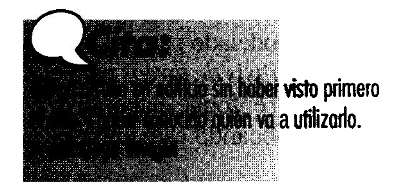


FIGURA 15.1. El proceso de diseño de la interfaz de usuario.

La espiral que se muestra en la Figura 15.1 implica que cada una de las tareas anteriores aparecerán más de una vez, en donde a medida que se avanza por la espiral se representará la elaboración adicional de los requisitos y el diseño resultante. En la mayoría de los casos, la actividad de implementación implica la generación de prototipos —la única forma práctica para validar lo que se **ha** diseñado—.

La actividad de análisis inicial se concentra en el perfil de los usuarios que van a interactuar con el sistema. Se registran el nivel de conocimiento, la comprensión del negocio **y** la receptividad general del nuevo sistema, y se definen diferentes categorías de usuarios. En cada categoría se lleva a cabo la elicitación de los requisitos. Esencialmente, el ingeniero del software intenta comprender la percepción del sistema (Sección 15.2.1) para cada clase de usuario.

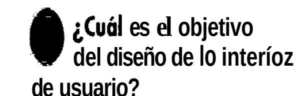


Una vez definidos los requisitos generales, se lleva a cabo un análisis más detallado de las tareas. Se identifican, describen y elaboran las tareas que lleva a cabo el usuario para conseguir los objetivos (por encima de la cantidad de pasos iterativos a través de la espiral). El análisis de tareas se estudia detalladamente en la Sección 15.3.

El análisis del entorno de usuario se centra en el entorno del trabajo físico. Entre las preguntas que se formulan se encuentran las siguientes:

- ¿Dónde se ubicará físicamente la interfaz?
- ¿Dónde se situará el usuario? ¿Llevará a cabo tareas no relacionadas con el interfaz?
- ¿Se adapta bien el hardware a las limitaciones de luz, espacio o ruido?

Esta recopilación de información que forma parte de la actividad de análisis se utiliza para crear un modelo de análisis para la interfaz. Mediante esta base de modelo se comienza la actividad de diseño.



El objetivo del diseño de la interfaz es definir un conjunto de objetos y acciones de interfaz (y sus representaciones en la pantalla) que posibiliten al usuario llevar a cabo todas las tareas definidas de forma que cumplan todos los objetivos de usabilidad definidos por el sistema. El diseño de la interfaz se aborda más detalladamente en la Sección 15.4

La actividad de implementación comienza normalmente con la creación de un prototipo que permita evaluar los escenarios de utilización. A medida que avanza el proceso de diseño iterativo, y para completar la construcción de la interfaz, se puede utilizar un kit de herramientas de usuario (Sección 15.5).

La validación se centra en: (1) la habilidad de la interfaz para implementar correctamente todas la tareas del usuario, para acoplar todas las variaciones de tareas, y para archivar todos los requisitos generales del usuario; (2) el grado de facilidad de utilización de la interfaz y de aprendizaje, y (3) la aceptación de la interfaz por parte del usuario como una herramienta Útil en su trabajo.

15.3 ANÁLISIS Y MODELADO DE TAREAS

En el Capítulo 13 se estudió la elaboración paso a paso (llamada también refinamiento paso a paso y descomposición funcional) como mecanismo para refinar las tareas de procesamiento necesarias para que el software lleve a cabo la función deseada. Más adelante en este mismo libro tendremos en consideración el análisis orientado a objetos como enfoque de modelado para los sistemas basados en computadora. El *análisis de tareas* para el diseño de la interfaz o bien utiliza un enfoque elaborativo o bien orientado a objetos, pero aplicando este enfoque a las actividades humanas.

El análisis de tareas se puede aplicar de dos maneras. Como ya hemos destacado anteriormente, un sistema interactivo basado en computadora se suele utilizar para reemplazar una actividad manual o semi-manual. Para comprender las tareas que se han de llevar a cabo para lograr el objetivo de la actividad, un ingeniero[4] deberá entender las tareas que realizan los hombres actualmente (cuando se utiliza un enfoque manual) y hacer corresponder esta tareas con un conjunto de tareas similar (aunque no necesariamente idénticas) que se implementan en el contexto de la interfaz de usuario. De forma alternativa, el ingeniero puede estudiar la especificación existente para la solución basada en computadora y extraer un conjunto de tareas que se ajusten al modelo de usuario, al modelo de diseño y a la percepción del sistema.

Independientemente del enfoque global utilizado para el análisis de tareas, el ingeniero deberá en primer lugar definirlas y clasificarlas. Ya se ha descrito anteriormente que el enfoque es una elaboración paso a paso. Por ejemplo, supongamos que una compañía pequeña quiere construir un sistema de diseño asistido por computadora explícitamente para diseñadores de interiores. Al observar a un diseñador de interiores en su trabajo, el ingeniero se da cuenta y notifica que el diseño interior se compone de una serie de actividades importantes: diseño del mobiliario, selección de tejidos y materiales, selección de decorados en paredes y ventanas, presentación (al cliente), costes y compras. Todas y cada una de estas tareas pueden elaborarse en otras subtareas. Por ejemplo, el diseño del mobiliario puede refinarse en las tareas siguientes: (1) dibujar el plano de la casa con las dimensiones de las habitaciones; (2) ubicar ventanas y puertas en los lugares adecuados; (3) utilizar plantillas de muebles para dibujar en el plano un esbozo del mobiliario a escala; **(3)** mover el esbozo del mobiliario para mejorar su colocación; *(5)* etiquetar el esbozo del mobiliario; (6) dibujar las dimensiones para mostrar la colocación; (7) realizar un dibujo en perspectiva para el cliente. Para las otras tareas importantes del enfoque se puede utilizar un método similar.



las tareas humanas se definen y se clasifican como parte del anólisis de las tareas. Para refinar los tareas se utiliza un proceso de elaboración. De forma alternativa, se identifican y se refinan los objetos y los acciones.

Las subtareas (1)-(7) pueden recibir más refinamiento. Las subtareas (1)-(6) se llevarán a cabo mediante la manipulación de información y mediante la realización de acciones dentro de la interfaz de usuario. Por otro lado, la subtarea (7) se podrá llevar a cabo automáticamente en el software y dará como resultado muy poca interacción directa con el usuario. El modelo de diseño de la interfaz deberá adaptarse a cada una de estas tareas de forma consecuente con el modelo del usuario (el perfil de un diseñador «típico» de interiores) y con la percepción del sistema (lo que el diseñador de interiores espera de un sistema automatizado).

15.4 ACTIVIDADES DEL DISEÑO DE LA INTERFAZ

Una vez finalizado el análisis de tareas, quedan definidas detalladamente todas (u objetos y acciones) las que requiere el usuario final y comienza la actividad del diseño de la interfaz. Mediante el enfoque que se muestra a continuación se podrán llevar a cabo los primeros pasos del diseño de la interfaz [NOR86]:

1. Establecer los objetivo? e intenciones para cada tarea.
2. Hacer corresponder cada objetivo/intención con una secuencia de acciones específicas
3. Especificar la secuencia de acciones de tareas y subtareas, también llamado *escenario del usuario,* de la manera en que se ejecutarán a nivel de la interfaz.
4. Indicar el estado del sistema, esto es, el aspecto que tiene la interfaz cuando se está llevando a cabo el escenario del usuario.
5. Definir los mecanismo de control, esto es, los objetos y acciones disponibles para que el usuario altere el estado del sistema.

¿Cuáles son los pasos *que* hay **que seguir** para **llevar** a **cabo el** diseño de **la interfaz?**

[4] En muchos casos las actividades descritas en esta sección son llevadas a cabo por un ingeniero del software. Esperemos que el individuo tenga experiencia en ingeniería humana y en el diseño de la interfaz de usuario.

[5] Entre los objetivos se pueden incluir la consideración de la utilidad de las tareas, la efectividad al llevar a cabo el objetivo comercial primordial, el grado de rapidez de aprendizaje de las tareas y el grado de satisfacción de los usuarios con la implementación final de la tarea.

6. Mostrar la forma en que los mecanismos de control afectan al estado del sistema.
7. Indicar la forma en que el usuario interpreta el estado del sistema a partir de la información proporcionada gracias a la interfaz.

Aunque el diseñador de la interfaz se guía por las reglas de oro abordadas en la Sección **15.1,** deberá considerar la forma en que se va a implementar la interfaz, el entorno (por ejemplo, tecnología de pantalla, sistema operativo, herramientas de desarrollo) que se va a utilizar y otros elementos de la aplicación que «se encuentren por detrás» de la interfaz.

15.4.1. Definición de objetos y acciones de la interfaz

Un paso importante en el diseño de la interfaz es la definición de los objetos y acciones que se van a aplicar. Para llevar a cabo esta definición, el escenario del usuario se analiza sintácticamente de manera muy similar a como se analizaban las narrativas de procesamiento del Capítulo 12. Esto es, se escribe la descripción del escenario de un usuario. Los sustantivos (objetos) y los verbos (acciones) se aíslan para crear una lista de objetos y de acciones.

Referencia cruzada

En la Sección 12.6.2 se **puede encontrar** un estudio completa del **análisis semántico gramatical**.

Una vez que se han definido y elaborado iterativamente tanto los objetos como las acciones, se establecen categorías por tipos. Los objetos se identifican como objetos origen, destino **y** de aplicación. Un *objeto origen* (por ejemplo, un icono de informes) se arrastra y se coloca sobre otro *objeto destino* (por ejemplo, un icono de impresora). La implicación de esta acción es crear una copia impresa de un informe. Un *objeto de aplicación* representa los datos específicos de la aplicación **que** no se manipulan directamente como parte de la interacción de la pantalla. Por ejemplo, una lista de correo postal se utiliza para almacenar los nombres que se utilizarán para un correo postal. Esta lista se puede ordenar, fusionar o purgar (acciones basadas en menú) pero **no** se puede arrastrar y colocar mediante la interacción del usuario.


¿Qué es el formato de pantalla y cómo se aplica?

Una vez que el diseñador queda satisfecho con la definición de todos los objetos y acciones importantes (para una iteración de diseño), se lleva a cabo el *formato de pantalla*. Al igual que otras actividades de diseño de la interfaz, el formato de pantalla es un proceso interactivo en donde se lleva a cabo el diseño gráfico **y** la colocación de los iconos, la definición del texto descriptivo en pantalla, la especificación y títulos para las ventanas, y la definición de los elementos del menú principales **y** secundarios. Si una metáfora con el mundo real es adecuada para la aplicación, queda especificada en ese momento y el formato se organiza para complementar esa metáfora.

Para mostrar la breve ilustración de los pasos de diseño descritos anteriormente, tomemos en consideración un escenario de usuario para la versión avanzada del sistema *HogarSeguro* (descrito en capítulos anteriores). En la versión avanzada, se puede acceder a *HogarSeguro* mediante módem *o* Intemet. Una aplicación para **PC** permite que un propietario compruebe el estado de la casa desde una localización remota, reinicializar la configuración *HogarSeguro,* activar y desactivar el sistema, (empleando una opción de vídeo con un coste extra[6]), y supervisar visualmente las habitaciones dentro de la casa. A continuación, se muestra un escenario preliminar de usuario para la interfaz:

Referencia cruzada

El escenario descrito aquí es similar a los casos de estudio descritos en el Capítulo 11.

Escenario. El propietario de la casa desea acceder al sistema *HogarSeguro* instalado en su casa. Mediante el sistema operativo de un PC remoto (por ejemplo, un portátil que el propietario se lleve al trabajo o de viaje), el propietario determina el estado del sistema de alarma, arma o desarma el sistema, reconfigura las zonas de seguridad y observa las diferentes habitaciones de la casa mediante la preinstalación de una cámara de vídeo.

Para acceder a *HogarSeguro* desde una localización remota, el propietario proporciona un identificador y una contraseña. Con esto se definen los niveles de acceso (por ejemplo, todos los usuarios puede que **no** tengan la capacidad de reconfigurar el sistema) y se proporciona seguridad. Una vez validados, el usuario (con **los** privilegios para el acceso) comprueba el estado del sistema, y cambia el estado armando y desarmando *HogarSeguro.* El usuario reconfigura el sistema visualizando todas las zonas configuradas actualmente, modificando las zonas cuando se requiera. El usuario observa el interior de la casa mediante cámaras de vídeo estratégicamente ubicadas. El usuario puede utilizar las cámaras para recorrer todo el interior y ampliarlo para ofrecer diferentes visiones del interior de la casa.

Tareas del propietario:

- *acceder* al sistema *HogarSeguro;*
- *introducir* un *ID y* una contraseña para permitir un acceso remoto;
- *comprobar* el estado del sistema;
- *activar* o *desactivar* el sistema *HogarSeguro;*

[6] La opción de vídeo posibilita al usuario colocar una cámara de vídeo en lugares clave por la casa y examinar la salida desde una localización remota. ¿El Gran Hermano?

- *visualizar* el plano **de** la *casa* **y** las l**ocalizaciones de** los **sensores;**
- *visualizar* zonas en el plano de la casa;
- *cambiar* zonas en el plano de la casa;
- *visualizar* las localizaciones **de** las cámaras **de vídeo** en el plano de la casa;
- *seleccionar* la cámara **de vídeo** para tener visión;
- *observar* las imágenes **de vídeo** (cuatro encuadres por segundo);
- *recorrer* y *ampliar* las habitaciones con la cámara **de vídeo.**

Los objetos (negrita) y las acciones (cursiva) se extraen de la lista de tareas del propietario descritas anteriormente. La mayoría de los objetos anteriores son objetos de aplicaciones. Sin embargo la localización de la cámara de vídeo (un objeto origen) se arrastra y se coloca sobre la cámara de vídeo (un objeto destino) para crear una imagen de vídeo (una ventana con la presentación de un vídeo).

Para la supervisión del vídeo se crea un esbozo del formato de pantalla (Fig. **15.2).** Para invocar la imagen de vídeo, se selecciona un icono de **localización de cámara de vídeo C,** ubicado en el **plano de la casa** que se visualiza en la **ventana de supervisión.** En este caso, la localización de una cámara en la sala de estar (SE), se arrastra y se coloca sobre el icono de vídeo de cámara en la parte superior izquierda de la pantalla. Entonces, mediante la visualización del recorrido realizado por el vídeo desde la cámara localizada en la sala de estar (SE), aparece la **ventana de imagen de vídeo.** Las diapositivas de control del **recorrido** por las habitaciones y de las **ampliaciones** se utilizan para controlar la amplitud y la dirección de la imagen de vídeo. Para seleccionar una visión desde otra cámara, el usuario simplemente arrastra y coloca el icono de **localización de cámara** dentro del icono **cámara** del ángulo superior izquierdo de la pantalla.

Esta muestra de esbozo de formato tendría que ir complementado mediante una ampliación de todos los elementos del menú dentro de la barra de menú, indicando las acciones disponibles para el (estado) *modo de supervisión del video.* Durante el diseño de la interfaz se debería crear un conjunto completo de esbozos para todas y cada una de las tareas del propietario descritas en el escenario del usuario.

15.4.2. Problemas del diseño

A medida que la interfaz de usuario va evolucionando casi siempre afloran cuatro temas comunes de diseño: el tiempo de respuesta del sistema, los servicios de ayuda al usuario, la manipulación de información de errores y el etiquetado de órdenes. Desgraciadamente, muchos diseñadores no abordan estos temas dentro del proceso de diseño hasta que es relativamente tarde (algunas veces no se siente la aparición de un error hasta que se dispone del prototipo operativo). El resultado suele ser una iteración innecesaria, demoras de proyecto y frustración del usuario. Es infinitamente mejor establecer el tema de diseño que se vaya a tener en cuenta al iniciar el diseño del software, es decir cuando los cambios son fáciles y los costes más reducidos.

Para muchas aplicaciones interactivas el tiempo de respuesta del sistema es el principal motivo de queja. En general, el tiempo de respuesta del sistema se mide desde el punto de vista que el usuario realiza la acción

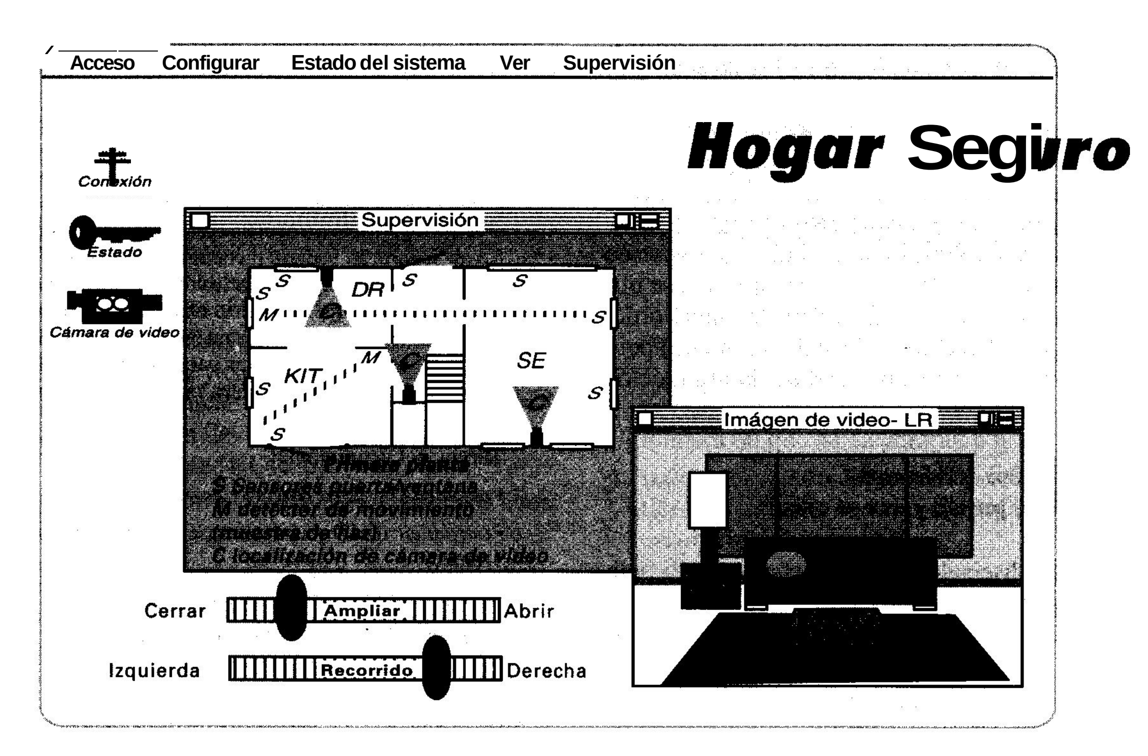


FIGURA 15.2. Formato preliminar de pantalla.

de control (por ejemplo, pulsar la tecla intro o pulsar el botón del ratón) hasta que el software responde con la salida o acción deseada.

El tiempo de respuesta del sistema tiene dos características importantes: la duración y la variabilidad. Si la *duración* de la respuesta del sistema es demasiado larga, es inevitable obtener como resultado la frustración y el estrés del usuario. Sin embargo, si la interfaz va marcando el ritmo del usuario una duración breve del tiempo de respuesta puede ser también perjudicial. Un tiempo de respuesta rápido puede obligar a que el usuario se precipite y cometa errores.



Si no se puede evitar una respuesta variable, asegúrese de proporcionar alguna indicación visual del progreso para que el usuario sepa lo que está ocurriendo.

La *variabilidad* se refiere a la desviación del tiempo de respuesta promedio, y en muchos aspectos es la característica **más** importante del tiempo de respuesta. Una variabilidad baja posibilita al usuario establecer un ritmo de interacción, aunque el tiempo de respuesta sea relativamente largo. Por ejemplo, es preferible obtener una segunda respuesta de una orden a una respuesta que varíe de 0,1 a 2,5 segundos. El usuario siempre estará desconcertado y preguntándose si ha ocurrido algo «diferente» detrás de la escena.

Casi todos **los** usuarios de un sistema interactivo basado en computadora requieren ayuda, ahora y siempre. Los dos tipos de funciones de ayuda más comunes son: integradas y complementarias (añadibles). [RUB88]. Se diseña una *función de ayuda integrada* dentro del mismo software desde el principio. Suele ser sensible al contexto, lo que posibilita al usuario seleccionar entre los temas que sean relevantes para las acciones que esté llevando a cabo en ese momento. Obviamente esto reduce el tiempo que requiere para obtener ayuda, e incrementa su «familiaridad» con la interfaz. *Una función de ayuda complementaria* se añade al software una vez construido el sistema. En muchos aspectos es muy similar a un manual de usuario en línea con una capacidad limitada de consulta. Es posible que el usuario tenga que buscar en una lista de miles de temas para encontrar la guía adecuada, entrando normalmente en las ayudas incorrectas y recibiendo mucha información irrelevante. No hay ninguna duda de que es preferible el enfoque de funciones de ayuda integradas al enfoque de funciones complementarias.

¿Cuáles son los temas de diseño que deberán tenerse en cuenta en la construcción de las funciones de ayuda?

Cuando se va a considerar un servicio de ayuda hay una serie de temas de diseño que deberán abordarse **[RUBSS]**:

- ¿Se necesitará disponer de todas las funciones del sistema en cualquier momento durante la interacción del sistema? Opciones: ayuda solo para un subconjunto de todas las funciones y acciones; ayuda para todas las funciones.
- ¿De qué forma solicitará ayuda el usuario? Opciones: un menú de ayuda; una tecla de función especial; una orden de AYUDA.
- ¿Cómo se representará la ayuda? Opciones: una ventana separada; una referencia a un documento impreso (no es lo ideal); una sugerencia de una o dos líneas que surge en una localización fija en la pantalla.
- ¿Cómo regresará el usuario a la interacción normal? Opciones: un botón de retorno visualizado en la pantalla; una tecla de función o una secuencia de control.
- ¿Cómo se estructurará la información sobre la pantalla? Opciones: una estructura «plana» donde el acceso a la información se realiza mediante una contraseña; una jerarquía estratificada de información que va proporcionando más datos a medida que el usuario va entrando por la estructura; la utilización de hipertexto.

Cuando ha salido algo mal, los mensajes de error y las sugerencias son «malas noticias» para los usuarios de sistemas interactivos. En el peor de los casos, estos mensajes imparten información sin utilizar o engañosa y sirven solo para incrementar la frustración del usuario. Existen muy pocos usuarios que puedan decir que no se han encontrado con un error del tipo:

```
FALLO GRAVE DEL SISTEMA -- 14A
```

En algún lugar debe existir una explicación del error 14A, o sino ¿por qué habrá incluido el diseñador esta identificación? A pesar de esto, el mensaje de error no proporciona una indicación verdadera de lo que va mal o de donde mirar para obtener más información. Un mensaje de error como el que se ha presentado anteriormente no hace nada por aliviar la ansiedad del usuario o por ayudar a solucionar el problema.

En general, todos los mensajes de error o sugerencias de un sistema interactivo deberán tener las características siguientes:

- El mensaje deberá describir el problema en una jerga que el usuario pueda entender.
- El mensaje deberá proporcionar consejos constructivos para recuperarse de un error.
- El mensaje deberá indicar cualquier consecuencia negativa del error (por ejemplo, los archivos de datos posiblemente deteriorados) para que el usuario pueda comprobar y garantizar que no hay ninguno (y corregirlos si existen).



Duplique el esfuerzo y las palabras al solucionar errores cuando piense que necesita una función de ayuda y, de esta forma, es probable que consiga un buen resultado.

- El mensaje deberá ir seguido por una clave audible o visual. Esto es, para acompañar la visualización del mensaje se podría generar un pitido, o aparecer momentáneamente una luz destelleante o visualizarse en un color que se pueda reconocer fácilmente como el «color del error».
- El mensaje no deberá tener «sentencias». Esto es, las palabras del mensaje nunca deberán culpar al usuario.

Dado que a nadie le gusta realmente tener malas noticias, a muy pocos usuarios les gustará tener un mensaje de error independientemente del diseño. No obstante, una filosofía eficaz de mensajes de error puede ayudar mucho a la hora de mejorar la calidad de un sistema interactivo y reducirá significativamente la frustración del usuario cuando aparecen problemas.

Anteriormente las órdenes escritas o tecleadas eran el modo de interacción más común entre el usuario y el software del sistema, y se utilizaban normalmente para aplicaciones de todo tipo. Actualmente, la utilización de interfaces orientadas a ventanas en donde solo se señala y se selecciona, ha reducido el hecho de depender de órdenes tecleadas, aunque muchos usuarios avanzados siguen prefiriendo el modo de interacción orientado a órdenes. Cuando se proporcionan Órdenes escritas como modo de interacción surge una serie de temas de diseño:

- ¿Se corresponden todas las opciones del menú con su órdenes?
- ¿Qué formas adquirirán las órdenes? Opciones: una secuencia de control (por ejemplo, alt-P); teclas de función; una palabra tecleada.
- ¿Será difícil aprender y recordar las órdenes? ¿Qué se puede hacer si se olvida una orden?
- ¿Podrá personalizar o abreviar el usuario las órdenes?

15.5 HERRAMIENTAS DE IMPLEMENTACIÓN

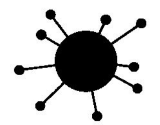


Una vez creado el modelo de diseño, se implementa como un prototipo[7], que los usuarios han examinado (aquellos que adaptan el modelo del usuario descrito anteriormente), y que se ha basado en los comentarios de los usuarios.

Para acoplar este enfoque de diseño iterativo se ha desarrollado una clase extensa de herramientas diseño de interfaz y de generación de prototipos. Estas herramientas así llamadas, *juego de herramientas de la interfaz de usuario o sistemas de desarrollo de la interfaz de usuario (SDIU)*, proporcionan componentes u objetos que facilitan la creación de ventanas, menús, interacción de dispositivos, mensajes de error, Órdenes y muchos otros elementos de un entorno interactivo.

Mediante los componentes de software preestablecidos que se pueden utilizar para crear una interfaz de usuario, un SDIU proporcionará los mecanismos [MYE89] para:

- gestionar los dispositivos de salida (tales como el ratón o el teclado);
- validar la entrada del usuario;
- manipular los errores y visualizar mensajes de error;
- proporcionar una respuesta (por ejemplo, un eco automático de la entrada)
- proporcionar ayuda e indicaciones de solicitud de entrada de órdenes;
- manipular ventanas y campos, desplazarse por las ventanas;
- establecer conexiones entre el software de la aplicación y la interfaz;
- aislar la aplicación de las funciones de gestión de la interfaz;
- permitir que el usuario personalice la interfaz.



Diseño de la Interfaz de usuario

Estas funciones se pueden implementar mediante un enfoque gráfico o basado en lenguajes.

15.6 EVALUACIÓN DEL DISEÑO

Una vez que se ha creado un prototipo de interfaz de usuario, deberá sufrir una evaluación para determinar si cumple las necesidades del usuario. La evaluación podrá abarcar un espectro de formalidad: desde «pruebas» informales en donde el usuario proporciona respuestas espontáneas hasta un estudio formalmente diseñado que utilizará métodos estadísticos para la evaluación de cuestionarios cumplimentados por un grupo de usuarios finales.

El ciclo de evaluación de la interfaz adquiere forma en la Figura 15.3. Una vez finalizado el modelo de dise-

[7] Debe destacarse que en algunos casos (por ejemplo, los indicadores del panel de los aviones), el primer paso debena ser simular la interfaz en un dispositivo en vez de utilizar el hardware del indicador.

ño, se crea un prototipo de primer nivel. Este prototipo es evaluado por el usuario, que es quien proporcionará al diseñador los comentarios directos sobre la eficacia de la interfaz. Además, si se utilizan técnicas formales de evaluación (por ejemplo, cuestionarios, hojas de evaluación), es posible que el diseñador extraiga información de estos datos (por ejemplo, el 80 por 100 de los usuarios no mostró afinidad con el mecanismo para grabar archivos de datos). Las modificaciones que se realicen sobre el diseño se basarán en la entrada del usuario y entonces se creará el prototipo de segundo nivel. El ciclo de evaluación continúa hasta que ya no sean necesarias más modificaciones del diseño de la interfaz.

Cita:

Un terminal no es una televisión vieja [illegible] máquina de escribir delante que pueda [illegible]. Es una interfaz en donde la mente [illegible] pueden conectar con el universo y los [illegible] por todos lados.

[illegible] Adams

El enfoque de generación de prototipos es eficaz, ahora bien ¿es posible evaluar la calidad de la interfaz de usuario antes de construir un prototipo? Si los problemas se pueden descubrir y solucionar rápidamente, el número de bucles en el ciclo de evaluación se reducirá y el tiempo de desarrollo se acortará. Si se ha creado un modelo de diseño de la interfaz, durante las primeras revisiones del diseño se podrán aplicar una serie de criterios [MOR81] de evaluación:

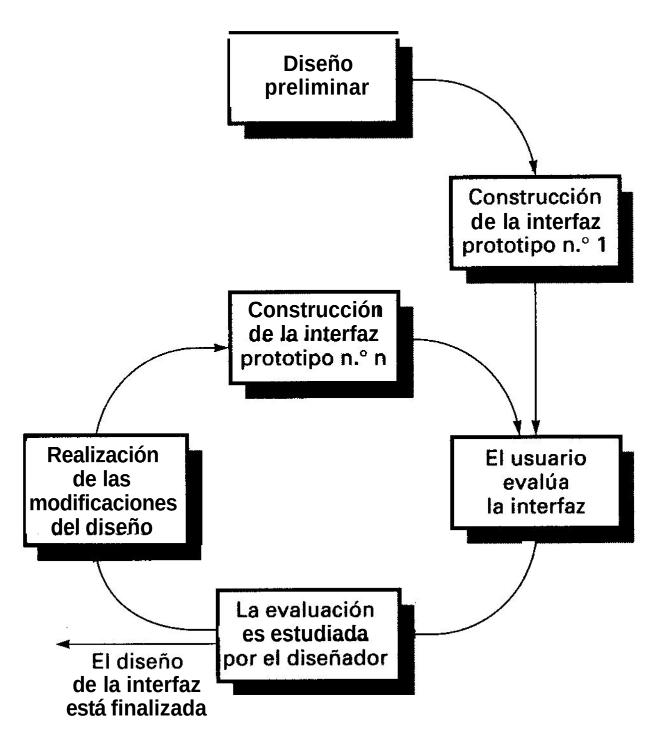


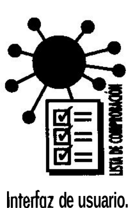
FIGURA 15.3. El ciclo de evaluación de diseño de la interfaz.

1. La duración y la complejidad de la especificación que se haya escrito del sistema y de su interfaz proporcionan una indicación de la cantidad de aprendizaje que requieren los usuarios del sistema.
2. La cantidad de **tareas** especificadas y la cantidad media de acciones por tarea proporcionan una indicación del tiempo y de la eficacia global del sistema.
3. La cantidad de acciones, tareas y estados de sistemas indicados con el modelo de diseño indican la carga de memoria que tienen los usuarios del sistema.
4. El estilo de la interfaz, las funciones de ayuda y el protocolo de solución de errores proporcionan una indicación general de la complejidad de la interfaz y el grado de aceptación por parte del usuario.

Una vez construido el primer prototipo, el diseñador puede recopilar una diversidad de datos cualitativos y cualificativos que ayudarán a evaluar la interfaz. Para recopilar los datos cualitativos, se pueden distribuir cuestionarios a los usuarios del prototipo. Las preguntas pueden ser (1) del tipo de respuesta si/no; (2) respuesta numérica; (3) respuesta con escala de valoración (subjetiva), o (4) respuesta por porcentajes (subjetiva). A continuación se muestran unos ejemplos:

1. ¿Eran claros los iconos? En caso negativo, ¿Qué iconos no eran claros?
2. ¿Eran fáciles de recordar y de invocar las acciones?
3. ¿Cuántas acciones diferentes ha utilizado?
4. ¿Resultaron fáciles de aprender las operaciones básicas del sistema? (valoración de 1 a 5)
5. En comparación con otras interfaces que haya utilizado, ¿Cómo evaluaría ésta?

entre el 1% mejores, 10% mejores, 25% mejores, 50% mejores, 50% inferiores.



Interfaz de usuario.

Si se desea obtener datos cuantitativos, se puede llevar a cabo una forma de análisis para el estudio del tiempo. **Los** usuarios son observados durante la interacción; y para tener una guía durante la modificación de la interacción se recopilan y utilizan datos tales como: grupo de tareas finalizadas correctamente por encima del período de tiempo estándar; frecuencia de acciones; secuencia de acciones; tiempo transcurrido «mirando» la pantalla; número y tipo de errores, y tiempo de solución de errores; tiempo necesario para utilizar la ayuda; y cantidad de referencias de ayuda por período de tiempo estándar.

Un estudio completo de **los** métodos de evaluación de la interfaz de usuario queda fuera del ámbito de este libro. Para más información, véase [LEA88] y [MAN97].

RESUMEN

Se puede argumentar que la interfaz de usuario es el elemento más importante de un sistema o producto basado en computadora. Si la interfaz tiene un diseño pobre, la capacidad que tiene el usuario de aprovecharse de la potencia de proceso de una aplicación se puede dificultar gravemente. En efecto, una interfaz débil puede llevar al fracaso de una aplicación con una implementación sólida y un buen diseño.

Existen tres principios importantes que dirigen el diseño de interfaces de usuario eficaces: (1) poner el control en manos del usuario; (2) reducir la carga de la memoria del usuario; **(3)** construir una interfaz consecuente. Para lograr que una interfaz se atenga a estos principios, se deberá desarrollar un proceso de diseño organizado.

El diseño de la interfaz de usuario comienza con la identificación de los requisitos del usuario, de las tareas y del entorno. El análisis de tareas es una actividad de diseño que define las tareas y acciones del usuario empleando un enfoque elaborativo u orientado a objetos.

Una vez que se han definido las tareas, los escenarios del usuario se crean y analizan para definir un conjunto de objetos y acciones de la interfaz. Esto es lo que proporciona la base para la creación del formato de la pantalla, el cual representa el diseño gráfico y la colocación de iconos, la definición de un texto descriptivo en pantalla, la especificación y titulación de ventanas y la especificación de los elementos importantes y secundarios del menú. Cuando se va a refinar un modelo de diseño para el sistema se tienen en consideración temas de diseño tales como tiempo de respuesta, estructura de Órdenes y acciones, manipulación de errores y funciones de ayuda. Para construir un prototipo que el usuario pueda evaluar se utilizan diversas herramientas de implementación.

La interfaz de usuario es la ventana del software. En muchos casos, la interfaz modela la percepción que tiene un usuario de la calidad del sistema. **Si** la ventana se difumina, se ondula o se quiebra, el usuario puede rechazar un sistema potente basado en computadora.

REFERENCIAS

[DUM88] Dumas, J.S., ***Designing User Interfaces for Software,*** Prentice-Hall, 1988.

[LEA88] Lea, M., «Evaluating User Interfaces Designs», ***User Interface Design for Cornputer Systems***, Halstead Press (Wiley), 1988.

[MAN97] Mandel, T., ***The Elements of User Interface Design,*** Wiley, 1997.

[MON84] Monk, A. (ed), ***Fundamentals of Human-Computer Interaction,*** Academic Press, 1984.

[MOR81] Moran, T.P., «The Command Language Grammar: A Representation for the User Interface **of** Interactive Computer Systems», ***Intl. Journal of Man-Machine Studies,*** **vol.** 15, pp. 3-50.

[MYE89] Myers, B.A., «User Interface Tools: Introduction and Survey», ***IEEE Software,*** Enero 1989, pp. 15-23.

[NOR86] Norman, D.A., «Cognitive Engineering», ***User Centered Systems Design,*** Lawrence Earlbaum Associates, Nueva Jersey, 1984.

[RUB88] Rubin, T., ***User Interface Design for Cornputer Systems,*** Haldstead Press (Wiley), 1988.

[SHN87] Shneiderman, B., ***Designing the User Interface,*** Addison-Wesley, 1987.

PROBLEMAS Y PUNTOS A CONSIDERAR

15.1. Describa la peor interfaz con la que haya trabajado alguna vez y critíquela en relación con los conceptos que se han presentado en este capítulo. Describa la mejor interfaz con la que haya trabajado alguna vez y critíquela en relación con los conceptos presentados en este capítulo.

15.2. Desarrolle dos principios de diseño más que «den el control al usuario».

15.3. Desarrolle dos principios de diseño más que «reduzcan la carga de memoria del usuario».

15.4. Desarrolle dos principios de diseño más que «ayuden a construir una interfaz consecuente».

15.5. Considere una de las aplicaciones interactivas siguientes (o una aplicación asignada por su profesor):

a. un sistema de autoedición
b. un sistema de diseño asistido por computadora
c. un sistema de diseño de interiores
d. un sistema de matriculación automatizado para la universidad
e. un sistema de gestión de biblioteca
f. un sistema de votación basada en Intemet para las elecciones públicas
g. un sistema bancario en casa
h. una aplicación interactiva asignada por su instructor

Desarrolle un modelo de diseño, un modelo de usuario, una imagen de sistema y una percepción de sistema para cualquiera de los sistemas anteriores.

15.6. Realice un análisis detallado de tareas para cualquiera de los sistemas que se relacionan en el Problema 15.5. Utilice un enfoque elaborativo u orientado a objetos.

15.7. Como continuación al Problema 15.6, defina objetos y acciones para la aplicación que acaba de seleccionar. Identifique todos los tipos de objetos.

15.8. Desarrolle un conjunto de formatos de pantalla con una definición de los elementos del menú principales y secundarios para el sistema que haya elegido en el Problema 15.5.

15.9. Desarrolle un conjunto de formatos de pantalla con una definición de los elementos principales y secundarios del menú para el sistema estándar *HogarSeguro* de la Sección 15.4.1. Puede optar por un enfoque diferente al mostrado en la Figura 15.2 sobre un formato de pantalla.

15.10. Describa el enfoque utilizado en las funciones de ayuda del usuario que haya utilizado para el modelo de diseño en el análisis de tareas que haya llevado a cabo desde el Problema 15.6 al 15.8.

15.11. Proporcione unos cuantos ejemplos que muestren la razón de que la variabilidad del tiempo de respuesta pueda considerarse un tema a tener en cuenta.

15.12. Desarrolle un enfoque que integre automáticamente los mensajes de error y una función de ayuda al usuario. Esto es, que el sistema reconozca automáticamente el tipo de error y proporcione una ventana de ayuda con sugerencias para corregirlo. Realice un diseño de software razonablemente completo que tenga en consideración estructuras de datos y algoritmos.

15.13. Desarrolle un cuestionario de evaluación de la interfaz que contenga 20 preguntas genéricas que se puedan aplicar a la mayoría de las interfaces. Haga que 10 compañeros de clase rellenen el cuestionario del sistema de interfaz que vayan a utilizar todos. Resuma los resultados e informe de ellos a la clase.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Aunque el libro de Donald Norman no trata específicamente interfaces hombre-máquina, mucho de lo que su libro ***(The Design of Everyday Things,*** Reissue edition, Currency/Doubleday, 1990) tiene que decir sobre la psicología de un diseño eficaz se aplicará a la interfaz de usuario. Es una lectura recomendada para todos los que sean serios a la hora de construir un diseño de interfaz de alta calidad.

Durante la década pasada se han escrito docenas de libros acerca del diseño de interfaces. Sin embargo, los libros de Mandel [MAN97] y Shneiderman ***(Designing the User Interface: Strategies for Effective Human-Computer Interaction,*** 3.ª ed., Addison-Wesley, 1990) continúan proporcionando el estudio más extenso acerca de la materia. Donnelly ***(In Your Face: The Best of Interactive Design,*** Rockport Publications, 1998), Fowler, Stanwick y Smith ***(GUI Design Handbook,*** McGraw-Hill, 1998), Weinschenk, Jamar, y Yeo ***(GUI Design Essentials,*** Wiley, 1997), Galitz ***(The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques,*** Wiley, 1996), Mullet y Sano ***(Designing Visual Interfaces: Communication Oriented Techniques,*** Prentice Hall, 1995), y Cooper ***(About Face: The Essentials of User Interface Design,*** IDG Books, 1995) han escrito estudios que proporcionan las líneas generales y principios de diseño adicionales así como las sugerencias para la elicitación de requisitos, modelado, implementación y comprobación del diseño.

El análisis y modelado de tareas son las actividades fundamentales del diseño de la interfaz. Hackos y Redish ***(User and Task Analysis for Interface Design,*** Wiley, 1998) han escrito un libro especializado en estos temas y proporcionan un método detallado para abordar el análisis de tareas. Wood ***(User Interface Design: Bridging the Gap from User Requirements to Design,*** CRC Press, 1997) tiene en consideración la actividad de análisis para interfaces y la transición hacia las tareas de diseño. Uno de los primeros libros que presentan el tema de los escenarios en el diseño de la interfaz de usuario ha sido editado por Carroll ***(Scenario-Based Design: Envisioning Work and Technology in System Development,*** Wiley, 1995). Horrocks ***(Constructing the User Interface with Statecharts,*** Addison-Wesley, 1998) ha desarrollado un método formal para el diseño de interfaces de usuario que se basa en el modelado del comportamiento basado en el estado.

La actividad de evaluación se centra en la usabilidad. Los libros de Rubin ***(Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests,*** Wiley, 1994) y Nielson ***(Usability Inspection Methods,*** Wiley, 1994) aborda el tema de forma considerable y detallada.

La computadora Apple de Macintosh popularizó las interfaces de usuario con diseños sólidos y fáciles de utilizar. El personal de Apple ***(Macintosh Human Interface Guidelines,*** Addison-Wesley, 1993) estudia la apariencia y utilización del ahora famoso Macintosh (y tan imitado). Uno de los primeros libros que se han escrito acerca de la interfaz de Microsoft Windows fue elaborado por el personal de Microsoft ***(The Windows Guidelines for Software Design: An Application Design Guide,*** Microsoft Press, 1995).

En un libro de Murphy ***(Front Panel: Designing Software for Embedded User Interfaces,*** R&D Books, 1998), el cual puede resultar de gran interés para los diseñadores del producto, se proporciona una guía detallada para el diseño de interfaces de sistemas empotrados y aborda los peligros inherentes en controles, en manipulación de maquinaria pesada e interfaces para sistemas médicos y de transporte. El diseño de la interfaz para productos empotrados también se estudia en el libro de Garrett ***(Advanced Instrumentation and Computer I/O Design: Real-Time System Computer Interface Engineering,*** IEEE, 1994).

Una amplia variedad de fuentes de información sobre el diseño de la interfaz de usuario y de temas relacionados están disponibles en Internet. Una lista actualizada de referencias relevantes para la interfaz de usuario se puede encontrar en **http://www.pressman5.com**

.

CAPÍTULO

16 DISEÑO A NIVEL DE COMPONENTES

EL diseño a nivel de componentes, llamado también *diseño procedimental,* tiene lugar después de haber establecido los diseños de datos, de interfaces y de arquitectura. El objetivo es convertir el modelo de diseño en un software operacional. No obstante, el nivel de abstracción del modelo de diseño existente es relativamente alto y el nivel de abstracción del programa operacional es bajo. Esta conversión puede ser un desafío, abriendo las puertas a la introducción de errores sutiles que sean difíciles de detectar y corregir en etapas posteriores del proceso del software. Edsgar Dijkstra, uno de los principales valedores para la comprensión del diseño, escribe lo siguiente [DIJ72]:

> El software parece ser diferente de muchos otros productos, donde como norma una calidad superior implica un precio más elevado. Aquellos que quieran realmente un software fiable descubrirán que para empezar deben encontrar un medio de evitar la mayoría de los errores y, como resultado, el proceso de programación será menos costoso.. ..y los programadores que sean eficaces no deberán malgastar su tiempo depurando los programas —no deberán cometer errores desde el principio—.

Aunque estas palabras ya se escribieron hace muchos años, hoy en día siguen siendo verdad. Cuando el modelo de diseño se convierte en código fuente, deberán seguirse una serie de principios que no solo lleven a cabo la conversión, sino que «no introduzcan errores desde el principio».

VISTAZO RÁPIDO

¿Qué es? Este diseño consiste en convertir el diseño de datos, interfaces y arquitectura en un software operacional. Para poderlo llevar a cabo, el diseño se deberá representar a un nivel de abstracción cercano a un código. El diseño a nivel de componentes establece los datos algorítmicos que se requieren para manipular las estructuras de datos, efectuar la comunicación entre los componentes del software por medio de las interfaces. e implementar los algoritmos asignados a cada componente

¿Quién lo hace? Un ingeniero del software.

¿Por qué es importante? Se tiene que tener la capacidad de determinar si el programa funcionará antes de construirlo. El diseño a nivel de componentes representa el software que permite revisar los datos del diseño para su corrección y consistencia con las representaciones de diseño anteriores (esto es, los diseño de datos, de interfaces y de arquitectura). Con este diseño se proporciona un medio de evaluar el funcionamiento de las estructuras de datos, interfaces y algoritmos.

¿Cuáles son los pasos? Las representaciones de los diseños de datos. arquitectura e interfaces forman la base del diseño a nivel de componentes. La narrativa del proceso de cada componente se convierte en un modelo de diseño procedimental empleando un conjunto de construcciones de programación estructurada. Para representar este diseño se utilizan las notaciones gráficas, tabulares y basadas en texto.

¿Cuál es el producto obtenido? El diseño procedimental de cada componente representado en forma de notación gráfica, tabular o basada en texto es el primer producto de trabajo durante el diseño a nivel de componentes.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Mediante una revisión estructurada y una inspección. El examen del diseño se realiza para determinar si las estructuras de los datos, las secuencias del proceso y las condiciones lógicas son correctas, y para ver si producirán la transformación de datos y control adecuados que se asignó al componente durante los primeros pasos del diseño.

Mediante la utilización de un lenguaje de programación es posible representar el diseño a nivel de componentes. En esencia, el programa se crea empleando como guía el modelo de diseño. Un enfoque alternativo es representar el diseño procedimental mediante la utilización de alguna representación intermedia (por ejemplo, gráfica, tabular o basada en texto) que se pueda transformar fácilmente en código fuente. Independientemente del mecanismo que se utilice para representar el diseño a nivel de componentes, la definición de las estructuras de datos, interfaces y algoritmos deberán ajustarse a la diversidad de líneas generales del diseño procedimental establecidas como ayuda para evitar errores durante la evolución del mismo diseño. En este capítulo, examinaremos estas líneas generales de diseño.

16.1 PROGRAMACIÓN ESTRUCTURADA

Los fundamentos del diseño a nivel de componentes proceden de la década de los años sesenta, y tomaron cuerpo con el trabajo de Edsgar Dijkstra y sus colaboradores [BOH66, DIJ65, DIJ76]. A finales de los sesenta, Dijkstra y otros propusieron la utilización de un conjunto de construcciones lógicas restringidas de las que poder formar cualquier programa.

Cita:
Cuando estoy trabajando en un problema nunca pienso en lo bonito que es. Solo pienso en cómo resolverlo. Pero cuando lo acabo, me doy cuenta de que no está bien si el resultado no es bonito.
R. Buckminster Fuller

Las construcciones son secuenciales, condicionales y repetitivas. La construcción *secuencial* implementa los pasos del proceso esenciales para la especificación de cualquier algoritmo. La *condicional* proporciona las funciones para procesos seleccionados a partir de una condición lógica y la *repetitiva* proporciona los bucles. Las tres construcciones son fundamentales para la *programación estructurada* —una técnica importante de diseño a nivel de componentes—.

Las construcciones estructuradas se propusieron para restringir el diseño procedimental del software a un número reducido de operaciones predecibles. La métrica de la complejidad (Capítulo 19) indica que la utilización de construcciones estructuradas reduce la complejidad del programa y, por tanto, mejora la capacidad de comprender, comprobar y mantener. La utilización de un número limitado de construcciones lógicas también contribuye a un proceso de comprensión humana que los psicólogos denominan *fragmentación* (troceado *o chunking)*. Para entender este proceso, consideremos la manera de leer esta página. Las letras no se leen individualmente: más bien, se reconocen formas o trozos de letras que forman palabras o frases. Las construcciones estructuradas son fragmentos lógicos que permiten al lector reconocer elementos procedimentales de un módulo en lugar de leer el diseño o el código línea a línea. La comprensión mejora cuando se encuentran patrones fácilmente reconocibles.

16.1.1. Notación gráfica del diseño

«Una imagen vale más que mil palabras», pero es importante saber qué imagen y qué 1.000 palabras. Es incuestionable que herramientas gráficas, tales como diagramas de flujo o diagramas de cajas, proporcionan formas gráficas excelentes que representan datos procedimentales fácilmente.



La programación estructurada proporciona los modelos lógicos y útiles para el diseñador.

El diagrama de flujo es una imagen bastante sencilla. Mediante la utilización de una caja se indica un paso del proceso. Un rombo representa una condición lógica y las flechas indican el flujo de control. La Figura 16.1 ilustra tres construcciones estructuradas. La secuencia se representa como dos cajas de procesamiento conectadas por una línea (flecha) de control. La condición, llamada también *si-entonces-si_no,* se representa

Primera tarea
Siguiente tarea
Secuencia
Condición
F
T
Parte entonces
Si entonces si-no (*if-then-else*)
Condición del caso
T
F
Parte caso
Según-sea (*selección*)
Tarea de bucle
Mientras-hacer (do-while)
Condición de bucle
Repetir-hasta (repeat-until)
Repetición

FIGURA 16.1. Construcciones en diagrama de flujo.

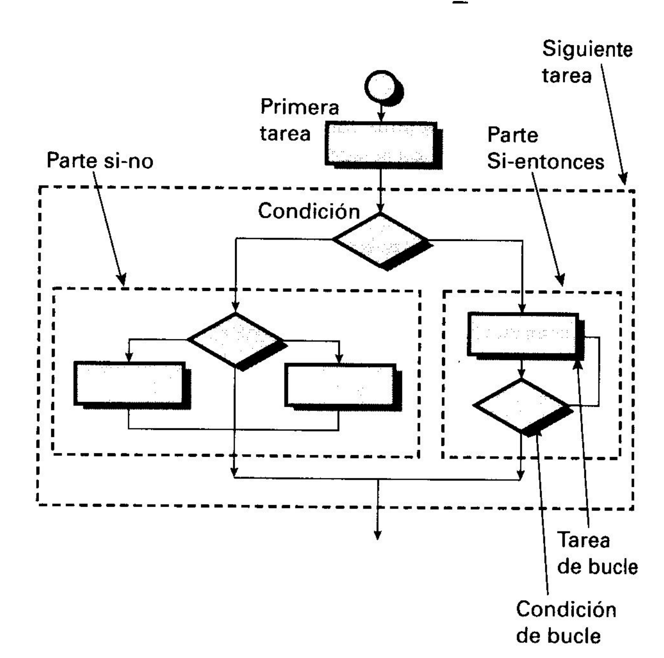


FIGURA 16.2. Construcciones anidadas.

mediante el símbolo del rombo de decisión que, si es cierto, provoca el procesamiento de la parte *entonces*, y, si es falso, invoca el procesamiento de la *parte si–no*. La repetición se representa mediante dos formas ligeramente diferentes. El *mientras-hacer* prueba una condición y ejecuta una tarea de bucle repetidamente siempre que la condición siga siendo verdad. Un *repetir-hasta* primero ejecuta la tarea de bucle, después prueba la condición y repite la tarea hasta que la condición falla. La construcción de selección (*según_sea*) que se muestra en la figura realmente es una extensión de *si-entonces-si_no*. Un parámetro se prueba por decisiones sucesivas hasta que ocurre una condición verdadera y se ejecuta el camino de procesamiento asociado.

Las construcciones estructuradas pueden anidarse unas en otras como muestra la Figura 16.2. En esta Figura, un repetir-hasta forma la parte then de un si-entonces-si–no (mostrada déntro de la línea discontinua exterior). Otro if-then-else forma la parte si-no de la primera condición. Finalmente la condición propiamente dicha se convierte en un segundo bloque en una secuencia. Anidando construcciones de esta manera, se puede desarrollar un esquema lógico complejo. Se debería destacar que cualquiera de los bloques de la Figura 16.2 podría hacer referencia a otro módulo, logrando por tanto la estratificación procedimental que conlleva la estructura del programa.



las construcciones de programación estructurado deberán facilitar la comprensión del diseño. Es correcto soltarlos, si utilizarlos sin «saltarlas» da como resultado uno complejidad innecesaria

En general, la utilización dogmática de construcciones estructuradas exclusivamente puede introducir ineficiencia cuando se requiere salir de un conjunto de bucles anidados o condiciones anidadas. Lo que es más importante, una complicación adicional de todas las pruebas lógicas a lo largo del camino de salida puede oscurecer el flujo de control del software, aumentar la posibilidad de error y tener un impacto negativo en su lectura y mantenimiento. ¿Qué podemos hacer?

'El diseñador dispone de dos opciones: (1) la representación procedimental se rediseña de manera que la «rama de escape» no sea necesaria en una posición anidada en el flujo de control; o (2) las construcciones estructuradas se salten de una manera controlada; esto es, se diseña una rama restringida fuera del flujo anidado. La opción 1 es obviamente el enfoque ideal, pero la opción 2 se puede implantar sin romper el espíritu de la programación estructurada [KNU75].

Otra herramienta de diseño gráfico, el *diagrama de cajas*, surgió del deseo de desarrollar una representación de diseño procedimental que no permitiera la violación de las construcciones estructuradas. Desarrollada por Nassi y Schneiderman [NAS73] y ampliada por Chapin [CHA74], los diagramas (también denominados *diagramas Nassi-Schneiderman, diagramas N-S o diagramas Chapin)* tienen las características siguientes: (1) dominio funcional (es decir, el alcance de una repetición o de un si-entonces-si–no) bien definido y claramente visible como representación gráfica; (2) la transferencia arbitraria de control es imposible; (3) el alcance de los datos locales y/o generales se puede determinar fácilmente y (4) la recursividad es fácil de representar.

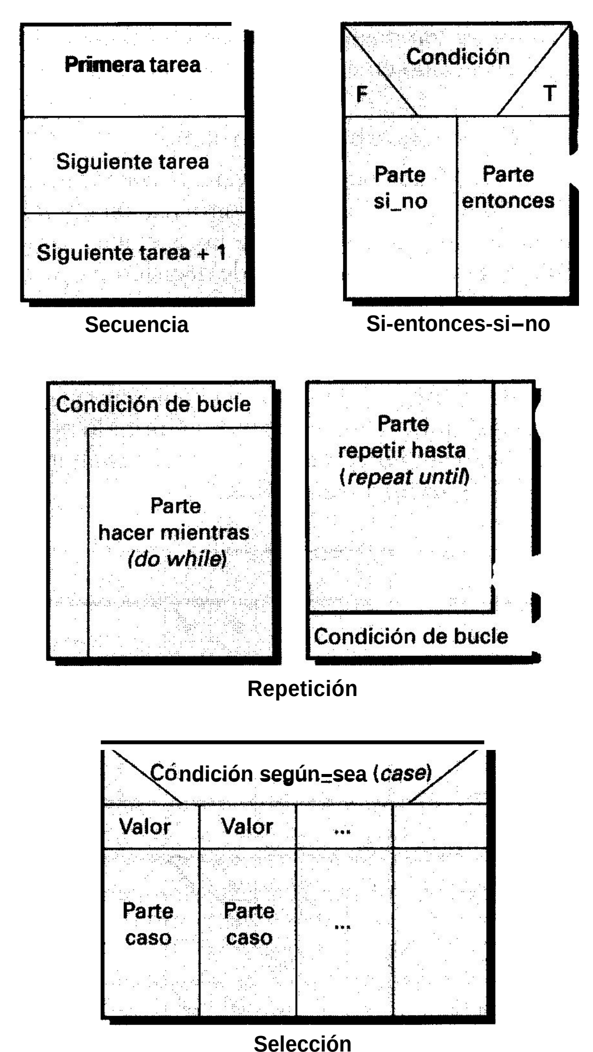


FIGURA 16.3. Construcciones de diagramas de capas.

La representación gráfica de construcciones estructuradas mediante diagramas de cajas se ilustra en la Figura 16.3. El elemento fundamental del diagrama es la caja. Para representar una secuencia, se conectan dos cajas seguidas. Para representar un si-entonces-si–no, la condición va seguida de una caja parte si-entonces y una parte sino. La repetición se dibuja con un límite que encierra el proceso (parte hacer-mientras o parte repetir-hasta) que se va a repetir. Finalmente, la selección se representa mediante la forma gráfica mostrada en la parte inferior de la figura.



Tonto los diagramas de flujo como los diagramas de cojos yo no se utilizan tonto como antes. En general, se deberían utilizar puro documentar o evaluar el diseño en casos específicos, no paro representar todo un sistema.

Al igual que los diagramas de flujo, un diagrama de cajas está estratificado en múltiples páginas a medida que se refinan los elementos de procesamiento de un módulo. Una «llamada» a un módulo subordinado se puede representar mediante una caja con el nombre del módulo encerrado en un óvalo.

16.1.2. Notación tabular de diseño

En muchas aplicaciones de software, puede ser necesario un módulo que evalúe una combinación compleja de condiciones y que seleccione las acciones basadas en esas condiciones. Las tablas de decisión proporcionan una notación que convierte acciones y condiciones (descritas en la narración del procesamiento) en una forma tabular. Es difícil que la tabla se pueda interpretar mal, e incluso es posible utilizarla como entrada legible por la máquina para un algoritmo dirigido por una tabla. En un estudio completo de esta herramienta de diseño, Ned Chapin afirma [HUR83]:

	Reglas								
Condiciones	1	2	3	4					n
condición n.º 1	✓			✓	✓				
condición n.º 2		✓		✓					
condición n.º 3			✓		✓				
Acciones									
acción n.º 1	✓			✓	✓				
acción n.º 2		✓		✓					
acción n.º 3			✓						
acción n.º 4			✓	✓	✓				
acción n.º 5	✓	✓			✓				

FIGURA 16.4. Nomenclatura de la tabla de decisión.

> Algunas herramientas de software antiguas se combinan bien con otras herramientas nuevas y técnicas de ingeniería del software. Las tablas de decisión son un ejemplo excelente. Fueron las que precedieron a la ingeniería del software hace casi una década, pero encajaron tan bien con la ingeniería del software que podrían haberse diseñado con tal propósito.

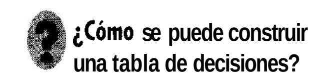


Utilice uno tabla de decisión cuando dentro de un componente se combine un conjunto completo de condiciones y de acciones.

En la Figura 16.4 se ilustra la organización de la tabla de decisión y se divide en cuatro secciones. El cuadrante superior izquierdo contiene una lista de todas las condiciones. El cuadrante inferior contiene una lista de todas las acciones posibles basándose en combinaciones de las condiciones. Los cuadrantes de la derecha forman una matriz que indica las combinaciones de las condiciones y las correspondientes acciones que se han de producir para cada combinación específica. Por tanto, cada columna de la matriz puede interpretarse como una *regla* de procesamiento.

Para desarrollar una tabla de decisión se aplican los siguientes pasos:

1. Hacer una lista de todas las acciones que pueden asociarse con un procesamiento específico (o módulo).
2. Hacer una lista de todas las condiciones (o decisiones) durante la ejecución del procesamiento.
3. Asociar conjuntos específicos de condiciones con acciones específicas, eliminando combinaciones imposibles de condiciones; alternativamente, desarrollar cualquier permutación posible de combinaciones.
4. Definir reglas indicando qué acción o acciones ocurren para un conjunto de condiciones.



Para ilustrar el empleo de una tabla de decisión tomemos en consideración el siguiente extracto de una descripción procedimental para un sistema de facturación de un servicio público:

> Si la cuenta del cliente se factura utilizando un método de tarifas fijo, se establece un cargo mensual mínimo para consumos menores de 100kWh (kilovatios/hora). En los demás casos, la facturación por computadora aplica la tarifa A. Sin embargo, si la cuenta se factura empleando un método de facturación variable, se aplicará la tarifa A para consumos por debajo de 100kWh, con un consumo adicional facturado de acuerdo con la tarifa B.

La Figura 16.5 ilustra una representación de tabla de decisión de la descripción anterior. Todas y cada una de las cinco reglas indican una condición de las cinco viables (esto es, en el contexto de este procedimiento una «V» (verdadera) no tiene sentido ni en la cuenta de tarifa fija ni en la variable, por tanto esta condición se omite). Como regla general, la tabla de decisión puede utilizarse eficazmente para complementar otras notaciones de diseño procedimental.

16.1.3. Lenguaje de diseño de programas

El *lenguaje de diseño de programas (LDP)*, también denominado *lenguaje estructurado* o *pseudocódigo*, es «un lenguaje rudimentario en el sentido de que utiliza el vocabulario de un lenguaje (por ejemplo, el Inglés), y la sintaxis global he otro (esto es, un lenguaje estruc-

turado de programación)» [CAI75]. En este capítulo se utiliza LDP como referencia genérica de un lenguaje de diseño.

A primera vista LDP se parece a un lenguaje de programación moderno. La diferencia entre éste y un verdadero lenguaje de programación radica en el empleo de texto descriptivo (por ejemplo, Inglés) insertado directamente dentro de las sentencias de LDP. Dado que se utiliza texto descriptivo insertado directamente en una estructura sintáctica, este lenguaje no se puede compilar (al menos por ahora). Sin embargo, las herramientas LDP que existen actualmente convierten LDP en un «esquema» de lenguaje de programación, y/o representación gráfica (por ejemplo, un diagrama de flujo de diseño). Estas herramientas también producen mapas anidados, un índice de operación de diseño, tablas de referencias cruzadas, y más diversidad de información.

	Reglas				
Condiciones	1	2	3	4	5
Cuenta de tarifa fija	V	V	F	F	F
Cuenta de tarifa variable	F	F	V	V	F
Consumo < 100kWh	V	F	V	F	
Consumo ≥ 100kWh	F	V	F	V	
Acciones					
Cargo mensual mínimo	✓				
Facturación tarifa A		✓	✓		
Facturación tarifa B				✓	
Otro tratamiento					✓

FIGURA 16.5. Tabla de decisión resultante.

Un lenguaje de diseño de programas puede ser una simple transposición de un lenguaje tal como Ada o C. Alternativamente, puede ser un producto comprado específicamente para el diseño procedimental.



Sería una buena idea utilizar un lenguaje de programación cama base para el LDP. Esto hará posible generar un esquema de código (mezclado can texto) a medida que se lleva a cabo el diseño a nivel de componentes.

Una sintaxis básica LDP deberá incluir construcciones para la definición de subprogramas, descripción de la interfaz, declaración de datos, técnicas para la estructuración de bloques, construcciones de condición, construcciones repetitivas y construcciones de E/S. El formato y la semántica de estas construcciones LDP se presentan en la sección siguiente.

Hay que destacar que LDP se puede ampliar de manera que incluya palabras clave para procesos multitarea y/o concurrentes, manipulación de interrupciones, sincronización entre procesos y muchas otras características. Los diseños de aplicaciones para los que se utiliza LDP deberán dictar la forma final que tendrá el lenguaje de diseño.

16.1.4. Un ejemplo de LDP

Para ilustrar el empleo de LDP presentamos un ejemplo de diseño procedimental para el software del sistema de seguridad ***HogarSeguro*** comentado en capítulos anteriores. El sistema ***HogarSeguro*** en cuestión vigila las alarmas de detección de fuego, humo, ladrones, agua y temperatura (por ejemplo, la ruptura del sistema de calefacción durante la ausencia del propietario en invierno); hace saltar el sonido de la alarma, y llama al servicio de vigilancia generando un mensaje de voz sintetizada. En el LDP que se muestra a continuación, se ilustran algunas de las construcciones importantes señaladas en secciones anteriores.

Hay que recordar que LDP no es un lenguaje de programación. El diseñador hace una adaptación cuando lo requiere y sin preocuparse de errores de sintaxis. Sin embargo, se deberá revisar el diseño del software de vigilancia (¿se observa algún problema?) y refinarlo antes de escribirse. El LDP siguiente define una elaboración de diseño procedimental para el componente de ***monitor de seguridad.***

```
PROCEDURE seguridad.monitor;
INTERFACE RETURNS sistema.estado;
TYPE señal IS STRUCTURE DEFINED
   nombre IS STRING LENGTH VAR;
   dirección IS HEX dispositivo localización;
   límite.valor IS superior límite SCALAR;
   mensaje IS STRING LENGTH VAR;
END señal TYPE;
TYPE sistema.estado IS BIT (4);
TYPE alarma.tipo DEFINED;
   humo.alarma IS INSTANCE OF señal;
   fuego.alarma IS INSTANCE OF señal;
   agua.alarma IS INSTANCE OF señal;
   temp.alarma IS INSTANCE OF señal;
   ladrón.alarma IS INSTANCE OF señal;
TYPE teléfono.número IS are código t 7 -dígito
número;



inicializar todos sistemas puertos y reinicializar
todo hardware;
CASE OF control.panel.interruptores (cps) ;
   WHEN cps = "test' SELECT
      CALL alarma PROCEDURE WITH "activado"para co
        probación. tiempo en segundos;
   WHEN cps = "alarma-desactivada"SELECT
      CALL alarma PROCEDURE WITH "desactivada";
```

```
    WHEN cps = "nuevo.límite.temp" SELECT
       CALL teclado.entrada PROCEDURE;
    WHEN cps = "ladrón alarma.desactivada" SELECT
       desactivar señal [ladrón.alarma];



    DEFAULT ninguno;
 ENDCASE
 REPEAT UNTIL activar.interruptor es desactivado
    reinicializar todos señal.valores e interruptores;
    DO FOR alarma.tipo = humo, fuego, agua, temp,
    ladrón;
       READ dirección[alarma.tipo] señal.valor;
       IF señal.valor > límite [alarma.tipo]
       THEN teléfono.mensaje = mensaje[alarma.tipo];
          establecer alarma.timbre en "activada"para
          alarma.tiemposegundos;
             PARBEGIN
             CALL alarma PROCEDURE WITH "activada",alar-
             ma.tiempo en segundos;
             CALL teléfono PROCEDURE WITH mensaje [alar-
             ma.tipo], teléfono número;
             ENDPAR
          ELSE bifurcación
          ENDIF
       ENDFOR
    ENDREP
    END seguridad.monitor
```

Hay que destacar que el diseñador del procedimiento del monitor de seguridad ha utilizado una construcción nueva ***PARBEGZN ... ENDPAR*** la cual especifica un bloque paralelo. Todas las tareas especificadas dentro del bloque ***PARBEGZN*** se ejecutan en paralelo. En este caso, no se toman en consideración los detalles de implementación.

16.2 COMPARACIÓN DE NOTACIONES DE DISEÑO

En la sección anterior se han presentado varias técnicas diferentes para representar un diseño procedimental. Se puede establecer una comparación teniendo la premisa de que cualquier notación para el diseño procedimental, si se utiliza correctamente, puede ser una ayuda incalculable para el proceso de diseño; por el contrario, aun con la mejor notación, si ésta se aplica mal, disminuye su entendimiento. Teniendo en cuenta lo anterior, es momento de examinar los criterios que se pueden aplicar para comparar las notaciones de diseño.

La notación de diseño deberá llevamos a una representación procedimental fácil de entender y de revisar. Además, la notación deberá mejorar la habilidad de «codificaren» para que el código se convierta de hecho en un subproducto natural de diseño. Finalmente, la representación de diseño deberá ser fácil de mantener para que el diseño sea siempre una representación correcta del programa.

¿Cuáles son los criterios que se utilizan para evaluar notaciones de diseño?

Dentro del contexto de las características generales que se describieron anteriormente se han establecido los siguientes atributos de notaciones de diseño:

Modularidad. Una notación de diseño deberá soportar el desarrollo del software modular y proporcionar un medio para la especificación de la interfaz.

Simplicidad general. Una notación de diseño deberá ser relativamente simple de aprender, relativamente fácil de utilizar y en general fácil de leer.

Facilidad de edición. Es posible que el diseño procedimental requiera alguna modificación a medida que el proceso de software avanza. La facilidad con la que un diseño se puede editar ayudará a simplificar todas y cada una de las tareas de la ingeniería del software.

Legibilidad para la computadora. Una notación que se puede introducir directamente en un sistema de desarrollo basado en computadora ofrece ventajas significativas.

Capacidad de mantenimiento. El mantenimiento del software es la fase más costosa del ciclo de vida del software. El mantenimiento de la configuración del software casi siempre lleva al mantenimiento de la representación del diseño procedimental.

Cumplimiento de las estructuras. Ya se han descrito las ventajas de un enfoque de diseño que utiliza conceptos de programación estructurada. Una notación de diseño que hace cumplir solo la utilización de construcciones estructuradas conduce a la práctica de un buen diseño.

Proceso automático. Un diseño procedimental contiene la información que se puede procesar para que el diseñador pueda observar otra vez y mejorar la corrección y calidad de un diseño. Dicha observación puede mejorarse con informes que provengan de herramientas de diseño de software.

Representación de datos. La habilidad de representar datos locales y globales es un elemento esencial del diseño detallado. Una notación de diseño ideal sería la representación directa de los datos.

Verificación de la lógica. La verificación automática de la lógica del diseño es el objetivo primordial durante las pruebas del software. Una notación que mejora la habilidad de verificar la lógica mejora enormemente lo aceptable de las pruebas.

Habilidad de «codificar en». La tarea de ingeniería del software que va a continuación del diseño a nivel de componentes es la generación de códigos. Una notación que puede convertirse fácilmente en código fuente reduce esfuerzos y errores.

Una pregunta que surge naturalmente de cualquier estudio de notaciones de diseño es: «¿Cuál es realmente la mejor notación según **los** atributos que se han establecido anteriormente?» La respuesta sería totalmente subjetiva **y** abierta a debate. Sin embargo, parece ser que el lenguaje de diseño de programas ofrece la mejor combinación de características. LDP puede insertarse directamente en listados de fuentes, en mejoras de documentación y al hacer que el mantenimiento del diseño sea menos difícil. La edición se puede llevar a cabo mediante cualquier editor de texto o sistema de procesamiento de texto; los procesadores automáticos ya existen, y el potencial de «generar códigos automáticos» es bueno.

Sin embargo, no se puede decir que la notación de diseños sea necesariamente inferior a LDP o que «sea buena» en atributos específicos. Muchos diseñadores prefieren la naturaleza pictórica de los diagramas de flujo y de los diagramas de bloques dado que proporcionan alguna perspectiva sobre el flujo de control. El contenido tabular preciso de las tablas de decisión es una herramienta excelente para las aplicaciones controladas por tablas. **Y** muchas otras representaciones de diseño (por ejemplo, véase [PET81], [SOM96]), que no se presentan en este libro, ofrecen **sus** propias ventajas. En el análisis final, la selección de una herramienta de diseño puede depender más de factores humanos [CUR85] que de atributos técnicos.

RESUMEN

El proceso de diseño acompaña a una secuencia de actividades que reducen lentamente el nivel de abstracción con el que se representa el software. El diseño a nivel de componentes representa el software a un nivel de abstracción muy cercano al código fuente.

A un nivel de componentes, el ingeniero del software debe representar estructuras de datos, interfaces **y** algoritmos con suficiente detalle como para servir de guía en la generación de códigos fuente de lenguajes de programación. Para conseguirlo, el ingeniero utiliza una de las notaciones de diseño que representa los detalles a nivel de componentes o bien en formatos gráficos, tabulares o basados en texto.

La programación estructurada es una filosofía de diseño procedimental que restringe el número y tipo de construcciones lógicas que se utilizan para representar el dato algorítmico. El objetivo de la programación estructurada es ayudar a que el diseñador defina algoritmos menos complejos y por tanto más fáciles de leer, comprobar y mantener.

REFERENCIAS

[BOH66] Bohm, C., y G. Jacopini, «Flow Diagramas, Turing Machines and Languages with only two Formation Rules», CACM, vol. 9, n.º 5, Mayo 1966, pp. 366-371.

[CAI75] Caine, S., y K. Gordon, «PDL —A Tool for Software Design», in ***Proc. National Computer Conference,*** AFIPS Press, 1975, pp. 271-276.

[CHA74] Chapin, N., «A New Format for Flowcharts», ***Software —Practice and Experience,*** vol. 4, n.º 4, 1974, pp. 341-357.

[DIJ65] Dijkstra, E., «Programming Considered as a Human Activity», in ***Proc. 1965 IFIP Congress,*** North Holland Publishing Co., 1965.

[DIJ72] Dijkstra, E., «The Humble Programmer», 1972 ACM Turing Award Lecture, ***CACM,*** vol. 15, n.º 10, Octubre 1972, pp. 859-866.

[DIJ76] Dijkstra, E., «Structure Programming», ***Software Engineering, Concepts and Techniques*** (J. Buxton et al., eds.), Van Nostrand-Reinhold, 1976.

[HUR83] Hurley, R.B., ***Decision Tables in Software Engineering,*** Van Nostrand Reinhold, 1983.

[LIN79] Linger, R.C, H. D. Mills y B.I. Witt, ***Estructured Programming,*** Addison-Wesley, 1979

[NAS73] Nassi, I., y B. Shneiderman, «Flowchart Techniques for Structured Programming», ***SIGPLAN Notices***, ACM, Agosto 1973.

[PET81] Peters, L.J., ***Software Design: Methods and Techniques,*** Yourdon Press, Nueva York, 1981.

[SOM96] Sommerville, I., ***Software Engineering,*** 5.ª ed., Addison-Wesley, 1996.

PROBLEMAS Y PUNTOS A CONSIDERAR

16.1. Seleccione una parte pequeña de un programa que ya exista (aproximadamente de 50-75 líneas de código). Aísle las construcciones de programación estructurada dibujando cajas alrededor de ellas en el código fuente. ¿Existen construcciones dentro del pasaje del programa que violen la filosofía de programación estructurada? Si fuera así, vuelva a diseñar el código para adaptarlo a las construcciones de programación estructurada. Si no fuera así, ¿Qué se puede destacar en las cajas que acaba de dibujar?

16.2. Todos los lenguajes de programación modernos implementan las construcciones de programación estructurada. Proporcione ejemplos de tres lenguajes de programación.

16.3. ¿Por qué es importante la «fragmentación» durante el proceso de revisión de diseño a nivel de componentes?

Los problemas 16.4-16.12 se pueden representar en una *(o* más) notaciones de diseño procedimentales presentadas en este capítulo. Su profesor puede asignar notaciones de diseño específicas a problemas concretos.

16.4. Desarrolle un diseño procedimental para los componentes que implementan las ordenaciones siguientes: Shell-Metzner; *heapsort* (ordenación del montículo). Si no está familiarizado con estas ordenaciones, consulte cualquier libro sobre estructuras de datos.

16.5. Desarrolle un diseño procedimental para una interfaz de usuario interactiva que solicite información básica acerca de los impuestos sobre la renta. Derive sus propios requisitos y suponga que todos los procesos sobre los impuestos son llevados a cabo por otros módulos.

16.6. Desarrolle un diseño procedimental para un programa que acepte un texto arbitrariamente largo como entrada y elabore una lista de palabras y de su frecuencia de aparición como salida.

16.7. Desarrolle un diseño procedimental de un programa que integre numéricamente una función *f* en los límites de a hacia ***b.***

16.8. Desarrolle un diseño procedimental para una máquina Turing generalizada que acepte un conjunto de cuádruples como entrada de programa y elabore una salida según especificación.

16.9. Desarrolle un diseño procedimental para un programa que solucione el problema de las Torres de Hanoi. Muchos libros de inteligencia artificial estudian este problema detalladamente.

16.10. Desarrolle un diseño procedimental para todas las partes o las partes fundamentales de un analizador sintáctico para un compilador. Consulte uno o más libros sobre diseño de compiladores.

16.11. Desarrolle un diseño procedimental para el algoritmo de encriptación/descriptación que haya seleccionado.

16.12. Escriba un argumento de una o dos páginas para la notación de diseño procedimental que prefiera. Asegúrese de que su argumento abarca los criterios presentados en la Sección 16.2.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

El trabajo de Linger, Mills y Witt *(Structured Programming-Theory and Practice,* Addison-Wesley, 1979) sigue siendo el estudio definitivo sobre esta materia. El texto contiene un buen LDP así como estudios detallados de ramificaciones de la programación estructurada. Entre otros libros que se centran en temas de diseño procedimental se incluyen los libros de Robertson *(Simple Program Design,* Boyd & Fraser Publishing, 1994), Bentley *(Programming Pearls,* Addison-Wesley, 1986, y *More Programming Pearls,* Addison-Wesley, 1988) y Dahl, Dijkstra y Hoare *(Structured Progamming,* Academic Press, 1972).

Solo existe un número relativamente pequeño de libros actuales dedicado únicamente al diseño a nivel de componentes. En general, los libros de lenguajes de programación abordan el diseño procedimental con algún detalle, pero siempre en el contexto del lenguaje que se ha introducido en este libro. Los siguientes libros son representativos de los cientos de títulos que tienen en consideración el diseño procedimental en un contexto de lenguaje de programación:

Adamson, T.A., K.C. Mansfield y J.L. Antonakos, *Structured Basic Applied to Technology,* Prentice Hall, *2000.*

Antonakos, J.L., y K. Mansfield, *Application Programming in Structured C,* Prentice Hall, 1996.

Forouzan, B.A., y R. Gilberg, *Computer Science: A Structured Programming Approach Using C++,* Brooks/Cole Publishing, 1999.

O'Brien, S.K., y S. Nameroff, *Turbo Pascal 7: The Complete Reference,* Osbome McGraw-Hill, 1993.

Welbum, T., y W. Price, *Structured Cobol: Fundamentals and Style,* 4.ª ed., Mitchell Publishers, 1995.

Una gran variedad de fuentes de información sobre diseño de software y temas relacionados están disponibles en Intemet. Una lista actualizada de referencias a sitios (lugares) web que son relevantes para conceptos y métodos de diseño se pueden encontrar en **http://www.pressman5.com**.

CAPÍTULO

17 TÉCNICAS DE PRUEBA DEL SOFTWARE

NUNCA se dará suficiente importancia a las pruebas del software y sus implicaciones en la calidad del software. Citando a Deutsch [DEU79]:

> El desarrollo de sistemas de software implica una serie de actividades de producción en las que las posibilidades de que aparezca el fallo humano son enormes. Los errores pueden empezar a darse desde el primer momento del proceso, en el que los objetivos....pueden estar especificados de forma errónea o imperfecta, así como [en] posteriores pasos de diseño y desarrollo...Debido a la imposibilidad humana de trabajar y comunicarse de forma perfecta, el desarrollo de software ha de ir acompañado de una actividad que garantice la calidad.

Las pruebas del software son un elemento crítico para la garantía de calidad del software y representa una revisión final de las especificaciones, del diseño y de la codificación.

La creciente percepción del software como un elemento del sistema y la importancia de los «costes» asociados a un fallo del propio sistema, están motivando la creación de pruebas minuciosas y bien planificadas. No es raro que una organización de desarrollo de software emplee entre el 30 y el 40 por ciento del esfuerzo total de un proyecto en las pruebas. En casos extremos, las pruebas del software para actividades críticas (por ejemplo, control de tráfico aéreo, control de reactores nucleares) puede costar ¡de tres a cinco veces más que el resto de los pasos de la ingeniería del software juntos!

VISTAZO RÁPIDO

¿Qué es? Una vez generado el código fuente, el software debe ser probado para descubrir (y corregir) el máximo de errores posibles antes de su entrega al cliente. Nuestro objetivo es diseñar una serie de casos de prueba que tengan una alta probabilidad de encontrar errores —pero ¿cómo conseguirlo?— Aquí es donde aplicamos las técnicas de pruebas del software. Estas técnicas facilitan una guía sistemática para diseñar pruebas que: (1) comprueben la lógica interna de los componentes software, y (2) verifiquen los dominios de entrada y salida del programa para descubrir errores en la funcionalidad, el comportamiento y rendimiento

¿Quién lo hace? Durante las primeras etapas de la prueba, es el ingeniero del software quien realiza todas las pruebas. Sin embargo, conforme progresa el proceso de prueba, los especialistas en pruebas se van incorporando.

¿Por qué es importante? Las revisiones y otras actividades SQA descubren errores, pero no son suficientes. Cada vez que el programa se ejecuta, el cliente lo está probando. Por lo tanto, debemos hacer un intento especial por encontrar y corregir todos los errores antes de entregar el programa al cliente. Con el objetivo de encontrar el mayor número posible de errores, las pruebas deben conducirse sistemáticamente. y los casos de prueba deben diseñarse utilizando técnicas definidas.

¿Cuáles son los pasos? El software debe probarse desde dos perspectivas diferentes: (1) la lógica interna del programa se comprueba utilizando técnicas de diseño de casos de prueba de «caja blanca». Los requisitos del software se comprueban utilizando técnicas de diseño de casos de prueba de «caja negra». En ambos casos, se intenta encontrar el mayor número de errores con la menor cantidad de esfuerzo y tiempo.

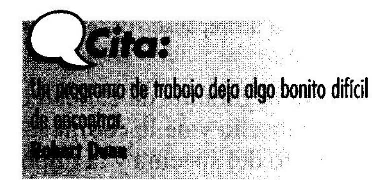
¿Cuál es el producto obtenido? Se define y documenta un conjunto de casos de prueba, diseñados para comprobar la lógica interna y los requisitos externos. Se determinan los resultados esperados y se guardan los resultados realmente obtenidos.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Cuando comienzas la prueba, debes cambiar tu punto de vista. Intenta «romper» con firmeza el software. Diseña casos de prueba de una forma disciplinada y revisa que dichos casos de prueba abarcan todo lo desarrollado.

En este capítulo, veremos los fundamentos de las pruebas del software y las técnicas de diseño de casos de prueba.

17.1 FUNDAMENTOS DE LAS PRUEBAS DEL SOFTWARE

Las pruebas presentan una interesante anomalía para el ingeniero del software. Durante las fases anteriores de definición y de desarrollo, el ingeniero intenta construir el software partiendo de un concepto abstracto y llegando a una implementación tangible. A continuación, llegan las pruebas. El ingeniero crea una serie de casos de prueba que intentan «demoler» el software construido. De hecho, las pruebas son uno de los pasos de la ingeniería del software que se puede ver (por lo menos, psicológicamente) como destructivo en lugar de constructivo.



Los ingenieros del software son, por naturaleza, personas constructivas. Las pruebas requieren que se descarten ideas preconcebidas sobre la «corrección» del software que se acaba de desarrollar y se supere cualquier conflicto de intereses que aparezcan cuando se descubran errores. Beizer [BEI90] describe eficientemente esta situación cuando plantea:

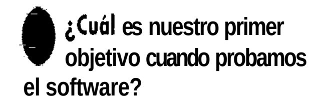
> Existe un mito que dice que si fuéramos realmente buenos programando, no habría errores que buscar. Si tan sólo fuéramos realmente capaces de concentrarnos, si todo el mundo empleara técnicas de codificación estructuradas, el diseño descendente, las tablas de decisión, si los programas se escribieran en un lenguaje apropiado, si tuviéramos siempre la solución más adecuada, entonces no habría errores. Así es el mito. Según el mito, existen errores porque somos malos en lo que hacemos, y si somos malos en lo que hacemos, deberíamos sentirnos culpables por ello. Por tanto, las pruebas, con el diseño de casos de prueba, son un reconocimiento de nuestros fallos, lo que implica una buena dosis de culpabilidad. Y lo tediosas que son las pruebas son un justo castigo a nuestros errores. ¿Castigados por qué? ¿Por ser humanos? ¿Culpables por qué? ¿Por no conseguir una perfección inhumana? ¿Por no poder distinguir entre lo que otro programador piensa y lo que dice? ¿Por no tener telepatía? ¿Por no resolver los problemas de comunicación humana que han estado presentes... durante cuarenta siglos?

¿Deben infundir culpabilidad las pruebas? ¿Son realmente destructivas? La respuesta a estas preguntas es: ¡No! Sin embargo, los objetivos de las pruebas son algo diferentes de lo que podríamos esperar.

17.1.1. Objetivos de las pruebas

En un excelente libro sobre las pruebas del software, Glen Myers [MYE79] establece varias normas que pueden servir acertadamente como objetivos de las pruebas:

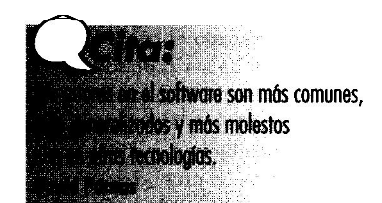
1. La prueba es el proceso de ejecución de un programa con la intención de descubrir un error.
2. Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.
3. Una prueba tiene éxito si descubre un error no detectado hasta entonces.


¿Cuál es nuestro primer objetivo cuando probamos el software?

Los objetivos anteriores suponen un cambio dramático de punto de vista. **Nos** quitan la idea que, normalmente, tenemos de que una prueba tiene éxito si no descubre errores.

Nuestro objetivo es diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y de esfuerzo.

Si la prueba se lleva a cabo con éxito (de acuerdo con el objetivo anteriormente establecido), descubrirá errores en el software. Como ventaja secundaria, la prueba demuestra hasta qué punto las funciones del software parecen funcionar de acuerdo con las especificaciones y parecen alcanzarse los requisitos de rendimiento. Además, los datos que se van recogiendo a medida que se lleva a cabo la prueba proporcionan una buena indicación de la fiabilidad del software y, de alguna manera, indican la calidad del software como un todo. Pero, la prueba no puede asegurar la ausencia de defectos; sólo puede demostrar que existen defectos en el software.



17.1.2. Principios de las pruebas

Antes de la aplicación de métodos para el diseño de casos de prueba efectivos, un ingeniero del software deberá entender los principios básicos que guían las pruebas del software. Davis [DAV95] sugiere un conjunto[1] de principios de prueba que se han adaptado para usarlos en este libro:

- ***A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente.*** Como hemos visto, el objetivo de las pruebas del software es descubrir errores. Se entiende que los defectos más graves (desde el punto de vista del cliente) son aquellos que impiden al programa cumplir sus requisitos.

[1] Aquí se presenta sólo un pequeño subconjunto de los principios de ingeniería de pruebas de Davis. Para obtener más información, vea [DAV96].

- *Las pruebas deberían planificarse mucho antes de que empiecen.* La planificación de las pruebas (Capítulo 18) pueden empezar tan pronto como esté completo el modelo de requisitos. La definición detallada de los casos de prueba puede empezar tan pronto como el modelo de diseño se ha consolidado. Por tanto, se pueden planificar y diseñar todas las pruebas antes de generar ningún código.
- *El principio de Pareto es aplicable a la prueba del software.* Dicho de manera sencilla, el principio de Pareto implica que al 80 por 100 de todos los errores descubiertos durante las pruebas se les puede hacer un seguimiento hasta un 20 por 100 de todos los módulos del programa. El problema, por supuesto, es aislar estos módulos sospechosos y probarlos concienzudamente.
- *Las pruebas deberían empezar por «lo pequeño» y progresar hacia «lo grande».* Las primeras pruebas planeadas y ejecutadas se centran generalmente en módulos individuales del programa. A medida que avanzan las pruebas, desplazan su punto de mira en un intento de encontrar errores en grupos integrados de módulos y finalmente en el sistema entero (Capítulo 18).
- *No son posibles las pruebas exhaustivas.* El número de permutaciones de caminos para incluso un programa de tamaño moderado es excepcionalmente grande (vea la Sección 17.2 para un estudio más avanzado). Por este motivo, es imposible ejecutar todas las combinaciones de caminos durante las pruebas. Es posible, sin embargo, cubrir adecuadamente la lógica del programa y asegurarse de que se han aplicado todas las condiciones en el diseño a nivel de componente.
- *Para ser más eficaces, las pruebas deberían ser realizadas por un equipo independiente.* Por «mas eficaces» queremos referirnos a pruebas con la más alta probabilidad de encontrar errores (el objetivo principal de las pruebas). Por las razones que se expusieron anteriormente en este capítulo, y que se estudian con más detalle en el Capítulo 18, el ingeniero del software que creó el sistema no es el más adecuado para llevar a cabo las pruebas para el software.

17.1.3. Facilidad de prueba

En circunstancias ideales, un ingeniero del software diseña un programa de computadora, un sistema o un producto con la «facilidad de prueba» en mente. Esto permite a los encargados de las pruebas diseñar casos de prueba más fácilmente. Pero, ¿qué es la «facilidad de prueba» James Bach[2] describe la facilidad de prueba de la siguiente manera:

La facilidad de prueba del software es simplemente la facilidad con la que se puede probar un programa de computadora. Como la prueba es tan profundamente difícil, merece la pena saber qué se puede hacer para hacerlo más sencillo. A veces los programadores están dispuestos a hacer cosas que faciliten el proceso de prueba y una lista de comprobación de posibles puntos de diseño, características, etc., puede ser útil a la hora de negociar con ellos.



Un útil documento titulado ((Perfeccionando la facilidad de la prueba del software» puede encontrarse en: **www.stlabs.com/newsletters/testnet/docs/testability.htm**.

Existen, de hecho, métricas que podrían usarse para medir la facilidad de prueba en la mayoría de sus aspectos. A veces, la facilidad de prueba se usa para indicar lo adecuadamente que un conjunto particular de pruebas va a cubrir un producto. También es empleada por los militares para indicar lo fácil que se puede comprobar y reparar una herramienta en plenas maniobras. Esos dos significados no son lo mismo que «facilidad de prueba del software». La siguiente lista de comprobación proporciona un conjunto de características que llevan a un software fácil de probar.

Operatividad. «Cuanto mejor funcione, más eficientemente se puede probar.»

- El sistema tiene pocos errores (los errores añaden sobrecarga de análisis y de generación de informes al proceso de prueba).
- Ningún error bloquea la ejecución de las pruebas
- El producto evoluciona en fases funcionales (permite simultanear el desarrollo y las pruebas).

Observabilidad. «Lo que ves es lo que pruebas.»

- Se genera una salida distinta para cada entrada.
- Los estados y variables del sistema están visibles o se pueden consultar durante la ejecución.
- Los estados y variables anteriores del sistema están visibles o se pueden consultar (por ejemplo, registros de transacción).
- Todos los factores que afectan a los resultados están visibles.
- Un resultado incorrecto se identifica fácilmente.
- Los errores internos se detectan automáticamente a través de mecanismos de auto-comprobación.
- Se informa automáticamente de los errores internos.
- El código fuente es accesible.

[2] Los siguientes párrafos son copyright de James Bach, 1994, y se han adaptado de una página de lnternet que apareció inicialmente en el grupo de noticias **comp.software-eng**. Este material se ha usado con permiso.



«La facilidad de prueba» ocurre como el resultado de un buen diseño. El diseño de datos, de la arquitectura, de las interfaces y de los componentes de detalle pueden facilitar la prueba o hacerlo más difícil.

Controlabilidad. «Cuanto mejor podamos controlar el software, más se puede automatizar y optimizar.»

- Todos los resultados posibles se pueden generar a través de alguna combinación de entrada.
- Todo el código es ejecutable a través de alguna combinación de entrada.
- El ingeniero de pruebas puede controlar directamente los estados y las variables del hardware y del software.
- Los formatos de las entradas y los resultados son consistentes y estructurados.
- Las pruebas pueden especificarse, automatizarse y reproducirse convenientemente.

Capacidad de descomposición. «Controlando el ámbito de las pruebas, podemos aislar más rápidamente los problemas y llevar a cabo mejores pruebas de regresión.»

- El sistema software está construido con módulos independientes.
- Los módulos del software se pueden probar independientemente

Simplicidad. «Cuanto menos haya que probar, más rápidamente podremos probarlo.»

- Simplicidad funcional (por ejemplo, el conjunto de características es el mínimo necesario para cumplir los requisitos).
- Simplicidad estructural (por ejemplo, la arquitectura es modularizada para limitar la propagación de fallos).
- Simplicidad del código (por ejemplo, se adopta un estándar de código para facilitar la inspección y el mantenimiento).

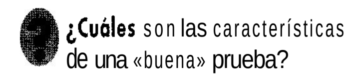
Estabilidad. «Cuanto menos cambios, menos interrupciones a las pruebas.»

- Los cambios del software son infrecuentes.
- Los cambios del software están controlados.
- Los cambios del software no invalidan las pruebas existentes.
- El software se recupera bien de los fallos.

Facilidad de comprensión. «Cuanta más información tengamos, más inteligentes serán las pruebas.»

- El diseño se ha entendido perfectamente.
- Las dependencias entre los componentes internos, externos y compartidos se han entendido perfectamente.
- Se han comunicado los cambias del diseño.
- La documentación técnica es instantáneamente accesible.
- La documentación técnica está bien organizada.
- La documentación técnica es específica y detallada.
- La documentación técnica es exacta.

Los atributos sugeridos por Bach los puede emplear el ingeniero del software para desarrollar una configuración del software (por ejemplo, programas, datos y documentos) que pueda probarse.



¿Y qué pasa con las pruebas propias? Kaner, Falk y Nguyen [KAN93] sugieren los siguientes atributos de una «buena» prueba:

1. Una buena prueba tiene una alta probabilidad de encontrar un error. Para alcanzar este objetivo, el responsable de la prueba debe entender el software e intentar desarrollar una imagen mental de cómo podría fallar el software.
2. Una buena prueba no debe ser redundante. El tiempo y los recursos para las pruebas son limitados. No hay motivo para realizar una prueba que tiene el mismo propósito que otra. Todas las pruebas debenan tener un propósito diferente (incluso si es sutilmente diferente). Por ejemplo, un módulo del software de *HogarSeguro* (estudiado en anteriores capítulos) está diseñado para reconocer una contraseña de usuario para activar o desactivar el sistema. En un esfuerzo por descubrir un error en la entrada de la contraseña, el responsable de la prueba diseña una serie de pruebas que introducen una secuencia de contraseñas. Se introducen contraseñas válidas y no válidas (secuencias de cuatro números) en pruebas separadas. Sin embargo, cada contraseña válida/no válida debería analizar un modo diferente de fallo. Por ejemplo, la contraseña no válida 1234 no debería ser aceptada por un sistema programado para reconocer 8080 como la contraseña correcta. Si 1234 es aceptada, tenemos presente un error. Otra prueba, digamos 1235, tendría el mismo propósito que 1234 y es, por tanto, redundante. Sin embargo, la entrada no válida 8081 u 8180 tiene una sutil diferencia, intentando demostrar que existe un error para las contraseñas «parecidas» pero no idénticas a la contraseña correcta.
3. Una buena prueba debería ser «la mejor de la cosecha» [KAN93]. En un grupo de pruebas que tienen propósito similar, las limitaciones de tiempo y recursos pueden abogar por la ejecución de sólo un subconjunto de estas pruebas. En tales casos, se debena emplear la prueba que tenga la más alta probabilidad de descubrir una clase entera de errores.
4. Una buena prueba no debería ser ni demasiado sencilla ni demasiado compleja. Aunque es posible a veces combinar una serie de pruebas en un caso de prueba, los posibles efectos secundarios de este enfoque pueden enmascarar errores. En general, cada prueba debería realizarse separadamente.

17.2. DISEÑO DE CASOS DE PRUEBA

El diseño de pruebas para el software o para otros productos de ingeniería puede requerir tanto esfuerzo como el propio diseño inicial del producto. Sin embargo, los ingenieros del software, por razones que ya hemos tratado, a menudo tratan las pruebas como algo sin importancia, desarrollando casos de prueba que «parezcan adecuados», pero que tienen poca garantía de ser completos. Recordando el objetivo de las pruebas, debemos diseñar pruebas que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y tiempo posible.



Existe una sola regla para el diseño de casos de prueba: cubrir todas las posibilidades, sin hacer demasiados casos de prueba.
Tsuneo Yamaura

Cualquier producto de ingeniería (y de muchos otros campos) puede probarse de una de estas dos formas: (1) conociendo la función específica para la que fue diseñado el producto, se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa y, al mismo, tiempo buscando errores en cada función; (2) conociendo el funcionamiento del producto, *se* pueden desarrollar pruebas que aseguren que «todas las piezas encajan», o sea, que la operación interna se ajusta a las especificaciones y que todos los componentes internos se han comprobado de forma adecuada. El primer enfoque de prueba se denomina prueba de caja negra y el segundo, prueba de caja blanca.



La página de técnicas de prueba es una excelente fuente de información sobre los métodos de prueba:
www.testworks.com/News/TTN-Online/

Cuando se considera el software de computadora, la *prueba de caja negra* se refiere a las pruebas que se llevan a cabo sobre la interfaz del software. O sea, los casos de prueba pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada y que se produce un resultado correcto, así como que la integridad de la información externa (por ejemplo, archivos de datos) se mantiene. Una prueba de caja negra examina algunos aspectos del modelo fundamental del sistema sin tener mucho en cuenta la estructura lógica interna del software.

La *prueba de caja blanca* del software se basa en el minucioso examen de los detalles procedimentales. Se comprueban los caminos lógicos del software proponiendo casos de prueba que ejerciten conjuntos específicos de condiciones y/o bucles. Se puede examinar el «estado del programa» en varios puntos para determinar si el estado real coincide con el esperado o mencionado.



Las pruebas de caja blanca son diseñadas después de que exista un diseño de componente (o código fuente). El detalle de la lógica del programa debe estar disponible.

A primera vista parecería que una prueba de caja blanca muy profunda nos llevaría a tener «programas cien por cien correctos». Todo lo que tenemos que hacer es definir todos los caminos lógicos, desarrollar casos de prueba que los ejerciten y evaluar los resultados, es decir, generar casos de prueba que ejerciten exhaustivamente la lógica del programa. Desgraciadamente, la prueba exhaustiva presenta ciertos problemas logísticos. Incluso para pequeños programas, el número de caminos lógicos posibles puede ser enorme. Por ejemplo, considere un programa de 100 líneas de código en lenguaje C. Después de la declaración de algunos datos básicos, el programa contiene dos bucles que se ejecutan de 1 a 20 veces cada uno, dependiendo de las condiciones especificadas en la entrada. Dentro del bucle interior, se necesitan cuatro construcciones *if-then-else*. ¡Existen aproximadamente 10^{14} caminos posibles que se pueden ejecutar en este programa!



No es posible una prueba exhaustiva sobre todos los caminos del programa, porque el número de caminos es simplemente demasiado grande.

Para poner de manifiesto el significado de este número, supongamos que hemos desarrollado un procesador de pruebas mágico («mágico» porque no existe tal procesador) para hacer una prueba exhaustiva. El procesador puede desarrollar un caso de prueba, ejecutarlo y evaluar los resultados en un milisegundo. Trabajando las 24 horas del día, 365 días al año, el procesador trabajaría durante 3170 años para probar el programa. Esto irremediablemente causaría estragos en la mayoría de los planes de desarrollo. La prueba exhaustiva es imposible para los grandes sistemas software.

La prueba de caja blanca, **sin** embargo, no se debe desechar como impracticable. Se pueden elegir y ejercitar una serie de caminos lógicos importantes.

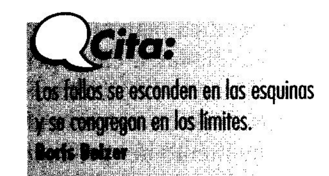
Se pueden comprobar las estructuras de datos más importantes para verificar **su** validez. Se pueden combinar los atributos de la prueba de caja blanca así como los de caja negra, para llegar a un método que valide la interfaz del software y asegure selectivamente que el funcionamiento interno del software es correcto.

17.3 PRUEBA DE CAJA BLANCA

La prueba de caja blanca, denominada a veces *prueba de caja de cristal* es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para obtener los casos de prueba. Mediante los métodos de prueba de caja blanca, el ingeniero del software puede obtener casos de prueba que **(1)** garanticen que se ejercita por lo menos una vez todos los caminos independientes de cada módulo; (2) ejerciten todas las decisiones lógicas en sus vertientes verdadera **y** falsa; **(3)** ejecuten todos los bucles en sus límites y con sus límites operacionales; y **(4)** ejerciten las estructuras internas de datos para asegurar su validez.

En este momento, se puede plantear un pregunta razonable: ¿Por qué emplear tiempo **y** energía preocupándose de (y probando) las minuciosidades lógicas cuando podríamos emplear mejor el esfuerzo asegurando que se han alcanzado los requisitos del programa? O, dicho de otra forma, ¿por qué no empleamos todas nuestras energías en la prueba de caja negra? La respuesta se encuentra en la naturaleza misma de los defectos del software (por ejemplo, [JON81]):

- Los errores lógicos **y** las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa. Los errores tienden a introducirse en nuestro trabajo cuando diseñamos e implementamos funciones, condiciones o controles que se encuentran fuera de lo normal. El procedimiento habitual tiende a hacerse más comprensible (y bien examinado), mientras que el procesamiento de «casos especiales» tiende a caer en el caos.
- **A** menudo creemos que un camino lógico tiene **pocas** posibilidades de ejecutarse cuando, de hecho, se puede ejecutar de forma normal. El flujo lógico de un programa a veces no es nada intuitivo, lo que significa que nuestras suposiciones intuitivas sobre el flujo de control **y** los datos nos pueden llevar a tener errores de diseño que sólo se descubren cuando comienza la prueba del camino.
- *Los errores tipográficos son aleatorios.* Cuando se traduce un programa a código fuente en un lenguaje de programación, es muy probable que se den algunos errores de escritura. Muchos serán descubiertos por los mecanismos de comprobación de sintaxis, pero otros permanecerán sin detectar hasta que comience la prueba. Es igual de probable que haya un error tipográfico en un oscuro camino lógico que en un camino principal.



17.4 PRUEBA DEL CAMINO BÁSICO

La *prueba del camino básico* es una técnica de prueba de caja blanca propuesta inicialmente por Tom McCabe [MCC76]. El método del camino básico permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental **y** usar esa medida como guía para la definición de un *conjunto básico* de caminos de ejecución. Los casos de prueba obtenidos del conjunto básico garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa.

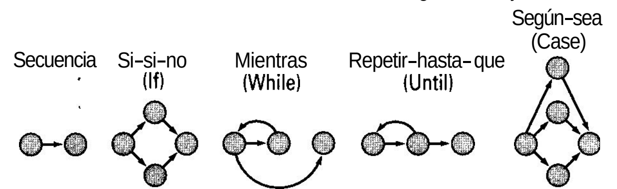


FIGURA 17.1. Notación de grafo de flujo.

17.4.1 Notación de grafo de flujo

Antes de considerar el método del camino básico se debe introducir una sencilla notación para la representación del flujo de control, denominada *grafo de flujo* (o *grafo del programa*)[3]. El grafo de flujo representa el flujo de control lógico mediante la notación ilustrada en la Figura 17.1. Cada construcción estructurada (Capítulo 16) tiene su correspondiente símbolo en el grafo del flujo.



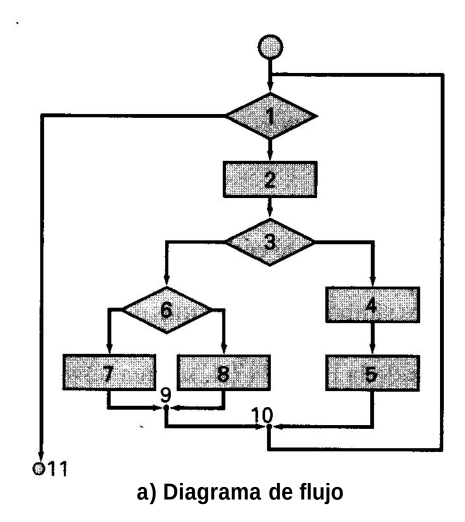
Dibujo un yrofo de flujo cuando lo lógica de lo estructura de control de un módulo seo complejo. El grafo de flujo te permite trazar más fácilmente los cominos del programa.

Para ilustrar el uso de un grafo de flujo, consideremos la representación del diseño procedimental en la Figura 17.2a. En ella, se usa un diagrama de flujo para

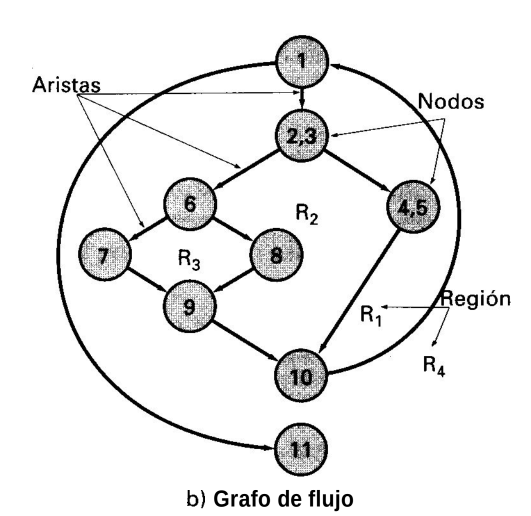
3 Realmente, el método del camino básico se puede llevar a cabo sin usar grafos de flujo. Sin embargo, sirve como herramienta Útil para ilustrar el método.

representar la estructura de control del programa. En la Figura 17.2b se muestra el grafo de flujo correspondiente al diagrama de flujo (suponiendo que no hay condiciones compuestas en los rombos de decisión del diagrama de flujo).

En la Figura 17.2b, cada círculo, denominado *nodo* del grafo de flujo, representa una o más sentencias procedimentales. Un solo nodo puede corresponder a una secuencia de *cuadros* de proceso y a un rombo de decisión. Las flechas del grafo de flujo, denominadas *aristas* o *enlaces*, representan flujo de control y son análogas a las flechas del diagrama de flujo. Una arista debe terminar en un nodo, incluso aunque el nodo no represente ninguna sentencia procedimental (por ejemplo, véase el símbolo para la construcción Si–entonces–si–no). Las áreas delimitadas por aristas y nodos se denominan *regiones*. Cuando contabilizamos las regiones incluimos el área exterior del grafo, contando como otra región más[4].



a) Diagrama de flujo



b) Grafo de flujo

FIGURA 17.2.

Cuando en un diseño procedimental se encuentran condiciones compuestas, la generación del grafo de flujo se hace un poco más complicada. Una condición compuesta se da cuando aparecen uno o más operadores (OR, AND, NAND, NOR lógicos) en una sentencia condicional. En la Figura 17.3 el segmento en LDP se traduce en el grafo de flujo anexo. Se crea un nodo aparte por cada una de las condiciones *a* y *b* de la sentencia **SI** *a* O *b*. Cada nodo que contiene una condición se denomina *nodo predicado* y está caracterizado porque dos o más aristas emergen de él.

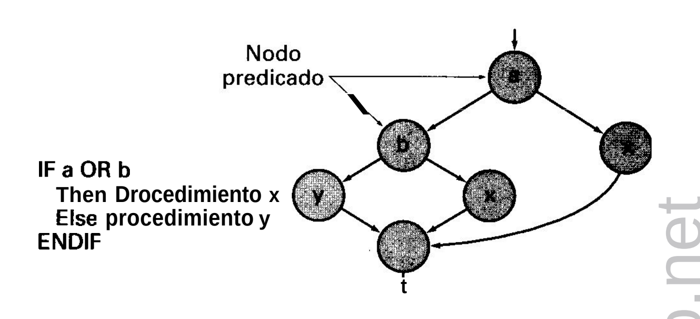


FIGURA 17.3. Lógica compuesta.

17.4.2. Complejidad ciclomática

La *complejidad ciclomática* es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa. Cuando se usa en el contexto del método de prueba del camino básico, el valor calculado como complejidad ciclomática define el número de *caminos independientes* del *conjunto básico* de un programa y nos da un límite superior para el número de pruebas que se deben realizar para asegurar que se ejecuta cada sentencia al menos una vez.

Un *camino independiente* es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una nueva condición. En términos del grafo de flujo, un camino independiente está constituido por lo menos por una arista que no haya sido recorrida anteriormente a la definición del camino. Por ejemplo, para el grafo de flujo de la Figura 17.2b, un conjunto de caminos independientes sería:

camino 1: 1-11
camino 2: 1-2-3-4-5-10-1-11
camino 3: 1-2-3-6-8-9-10-1-11
camino 4: 1-2-3-6-7-9-10-1-11

Fíjese que cada nuevo camino introduce una nueva arista. El camino

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

no se considera un camino independiente, ya que es simplemente una combinación de caminos ya especificados y no recorre ninguna nueva arista.

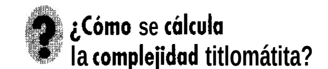
[4] En la Sección 17.6.1 viene un estudio mas detallado de los grafos y su empleo en las pruebas.



*La **complejidad ciclomática** es una métrica útil para predecir las módulos **que** san **más** propensas a error. Puede ser usada **tanto** para planificar pruebas cama **para** diseñar casas de prueba.*

Los caminos 1,2, 3 **y** 4 definidos anteriormente componen un *conjunto básico* para el grafo de flujo de la Figura 17.2b. Es decir, si se pueden diseñar pruebas que fuercen la ejecución de esos caminos (un conjunto básico), se garantizará que se habrá ejecutado al menos una vez cada sentencia del programa y que cada condición se habrá ejecutado en sus vertientes verdadera y falsa.

Se debe hacer hincapié en que el conjunto básico no es Único. De hecho, de un mismo diseño procedimental se pueden obtener varios conjuntos básicos diferentes.



¿Cómo sabemos cuántos caminos hemos de buscar? El cálculo de la complejidad ciclomática nos da la respuesta. La complejidad ciclomática está basada en la teoría de grafos y nos da una métrica del software extremadamente Útil. La complejidad se puede calcular de tres formas:

1. El número de regiones del grafo de flujo coincide con la complejidad ciclomática.
2. La complejidad ciclomática, $V(G)$, de un grafo de flujo G se define como.

$$V(G) = \mathbf{A} - N + 2$$

donde A es el número de aristas del grafo de flujo y N es el número de nodos del mismo.

3. La complejidad ciclomática, $V(G)$, de un grafo de flujo G también se define como

$$V(G) = P + 1$$

donde P es el número de nodos predicado contenidos en el grafo de flujo G.



La complejidad **ciclomática determina** el número de casos de prueba que deben **ejecutarse** para garantizar que **todas** las **sentencias** de un **componente** han sido **ejecutadas al** menos una vez.

Refiriéndonos de nuevo al grafo de flujo de la Figura 17.2b, la complejidad ciclomática se puede calcular mediante cualquiera de los anteriores algoritmos:

1. el grafo de flujo tiene cuatro regiones
2. V(G) = 11 aristas - 9 nodos + 2 = 4
3. V(G) = 3 nodos predicado + 1 = 4.

Por tanto, la complejidad ciclomática del grafo de flujo de la Figura 17.2b es 4.

Es más, el valor de $V(G)$ nos da un límite superior para el número de caminos independientes que componen el conjunto básico y, consecuentemente, un valor límite superior para el número de pruebas que se deben diseñar y ejecutar para garantizar que se cubren todas las sentencias del programa.

17.4.3. Obtención de casos de prueba

El método de prueba de camino básico se puede aplicar a un diseño procedimental detallado o a un código fuente. En esta sección, presentaremos la prueba del camino básico como una serie de pasos. Usaremos el procedimiento **media,** representado en LDP en la Figura 17.4, como ejemplo para ilustrar todos los pasos del método de diseño de casos de prueba. Se puede ver que **media,** aunque con un algoritmo extremadamente simple, contiene condiciones compuestas **y** bucles.



1. **Usando el diseño o el código como base, dibujamos el correspondiente grafo de flujo.** Creamos un grafo usando los símbolos y las normas de construcción presentadas en la Sección 16.4.1. Refiriéndonos al LDP de **media** de la Figura 17.4, se crea un grafo de flujo numerando las sentencias de LDP que aparecerán en los correspondientes nodos del grafo de flujo. El correspondiente grafo de flujo aparece en la Figura 17.5.

```
PROCEDURE media;
   * Este procedimiento calcula la media de 100 o menos
     números que se encuentren entre unos limites;
     también calcula el total de entradas y el total
     de números válidos.

     INFERFACE RETURNS media, total. entrada, total. válido;
     INTERFACE ACEPTS valor, mínimo, máximo;

     TYPE valor [1:100] IS SCALAR ARRAY;
     TYPE media, total. entrada, total. válido;
        Mínimo, máximo, suma IS SCALAR:
     TYPE i IS INTEGER;
     i = 1;
 1   total, entrada =total, válido = 0;            2
     suma =0;
     DO WHILE VALOR [i] < > -999 and total entrada < 100   3
 4 ----> Incrementar total entrada en 1;
         IF valor [i] > = mínimo  AND  valor [i] < = maximo   6
 5           THEN incrementar total.válido en 1;
          7       suma = suma +valor [i];
             ELSE ignorar
        8  ENDIF
           Incrementar i en 1;
    9  ENODO
       If total valido > 0    10
          THEN media = suma/total valido,
 12 ----> ELSE media = -999,
    13 ENDIF
END media
```

FIGURA 17.4. LDP para diseño de pruebas con nodos no identificados.

2. **Determinamos la complejidad ciclomática del grafo de flujo resultante.** La complejidad ciclomática, *V(G),* se determina aplicando uno de los algoritmos descritos en la Sección 17.5.2.. Se debe tener en cuenta que se puede determinar *V(G)* sin desarrollar el grafo de flujo, contando todas las sentencias condicionales del LDP (para el procedimiento **media** las condiciones compuestas cuentan como 2) **y** añadiendo 1.

   En la Figura 17.5,

   V(G) = 6 regiones

   V(G) = 17 aristas – 13 nodos +2 = 6

   V(G) = 5 nodos predicado + 1 = 6

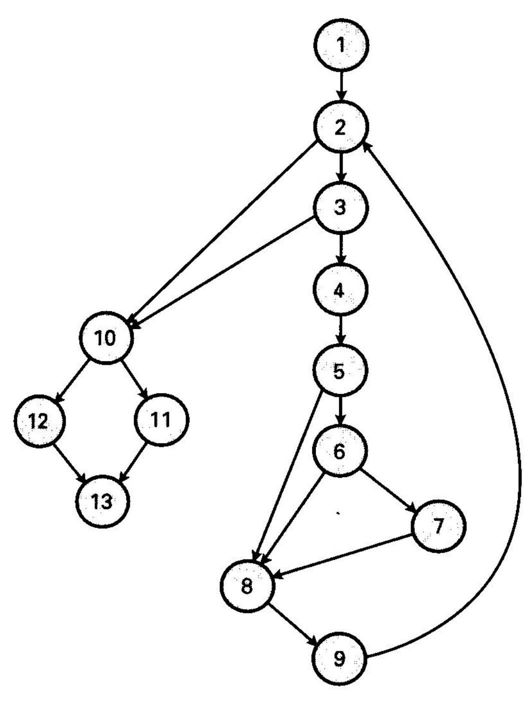


FIGURA 17.5. Grafo de flujo del procedimiento media.

3. **Determinamos un conjunto básico de caminos linealmente independientes.** El valor de V(G) nos da el número de caminos linealmente independientes de la estructura de control del programa. En el caso del procedimiento **media,** hay que especificar seis caminos:

   camino 1: 1-2-10-11-13

   camino 2: 1-2-10-12-13

   camino 3: 1-2-3-10-11-13

   camino 4: 1-2-3-4-5-8-9-2-...

   camino 5: 1-2-3-4-5-6-8-9-2-...

   camino 6: 1-2-3-4-5-6-7-8-9-2-...

   Los puntos suspensivos (...) que siguen a los caminos 4, 5 y 6 indican que cualquier camino del resto de la estructura de control es aceptable. Normalmente merece la pena identificar los nodos predicado para que sea más fácil obtener los casos de prueba. En este caso, los nodos 2, 3, 5, 6 **y** 10 son nodos predicado.

4. **Preparamos los casos de prueba que forzarán la ejecución de cada camino del conjunto básico.** Debemos escoger los datos de forma que las condiciones de los nodos predicado estén adecuadamente establecidas, con el fin de comprobar cada camino. Los casos de prueba que satisfacen el conjunto básico previamente descrito son:

Caso de prueba del camino 1:

valor ***(k)*** = entrada válida, donde ***k*** < i definida a continuación

valor (*i*) = –999, donde 2 <= i <= 100

resultados esperados: media correcta sobre los ***k*** valores y totales adecuados.

Nota: el camino 1 no se puede probar por **sí** solo; debe ser probado como parte de las pruebas de los caminos 4, 5 y 6.



Los ingenieros del software subestiman bastante el número de pruebas necesarias para verificar un programa simple.

Martyn Ould y Charles Unwin

Caso de prueba del camino 2:

valor (1) = –999

resultados esperados: media = –999; otros totales con sus valores iniciales

Caso de prueba del camino 3:

intento de procesar 101 *o* más valores

los primeros 100 valores deben ser válidos

resultados esperados: igual que en el caso de prueba 1

Caso de prueba del camino 4:

valor (*i*) = entrada válida donde i < 100

valor ***(k)*** < mínimo, para ***k*** < i

resultados esperados: media correcta sobre los **k** valores y totales adecuados

Caso de prueba del camino 5:

valor (*i*) = entrada válida donde i < 100

valor ***(k)*** > máximo, para ***k*** <= i

resultados esperados: media correcta sobre los *n* valores y totales adecuados

Caso de prueba del camino 6:

valor (*i*) = entrada válida donde i < 100

resultados esperados: media correcta sobre los *n* valores y totales adecuados

Ejecutamos cada caso de prueba y comparamos los resultados obtenidos con los esperados. Una vez terminados todos los casos de prueba, el responsable de la prueba podrá estar seguro de que todas las sentencias del programa se han ejecutado por lo menos una vez.

Es importante darse cuenta de que algunos caminos independientes (por ejemplo, el camino 1 de nuestro ejemplo) no se pueden probar de forma aislada. Es decir, la combinación de datos requerida para recorrer el camino no se puede conseguir con el flujo normal del programa. En tales casos, estos caminos se han de probar como parte de otra prueba de camino.

17.4.4. Matrices de grafos

El procedimiento para obtener el grafo de flujo, e incluso la determinación de un conjunto de caminos básicos, es susceptible de ser mecanizado. Para desarrollar una herramienta software que ayude en la prueba del camino básico, puede ser bastante Útil una estructura de datos denominada *matriz de grafo.*

Una matriz de grafo es una matriz cuadrada cuyo tamaño (es decir, el número de filas y de columnas) es igual al número de nodos del grafo de flujo. Cada fila y cada columna corresponde a un nodo específico y las entradas de la matriz corresponden a las *conexiones* (aristas) entre los nodos. En la Figura 17.6 se muestra un sencillo ejemplo de un grafo de flujo con su correspondiente matriz de grafo [BEI90].

En la figura, cada nodo del grafo de flujo está identificado por un número, mientras que cada arista lo está por su letra. Se sitúa una entrada en la matriz por cada conexión entre dos nodos. Por ejemplo, el nodo 3 está conectado con el nodo 4 por la arista b.

¿Qué es una matriz de grafos y cómo aplicarla en la prueba?

Hasta aquí, la matriz de grafo no es nada más que una representación tabular del grafo de flujo. Sin embargo, añadiendo un *peso de enlace* a cada entrada de la matriz, la matriz de grafo se puede convertir en una potente herramienta para la evaluación de la estructura de control del programa durante la prueba. El peso de enlace nos da información adicional sobre el flujo de control. En su forma más Sencilla, el peso de enlace es 1 (existe una conexión) ó 0 (no existe conexión). A los pesos de enlace se les puede asignar otras propiedades más interesantes:

- la probabilidad de que un enlace (arista) sea ejecutado;
- el tiempo de procesamiento asociado al recorrido de un enlace;
- la memoria requerida durante el recorrido de un enlace; y
- los recursos requeridos durante el recorrido de un enlace.

Para ilustrarlo, usaremos la forma más simple de peso, que indica la existencia de conexiones (0 ó 1). La matriz de grafo de la Figura 17.6 se rehace tal y como se muestra en la Figura 17.7. Se ha reemplazado cada letra por un 1, indicando la existencia de una conexión (se han excluido los ceros por claridad). Cuando se representa de esta forma, la matriz se denomina ***matriz de conexiones.***

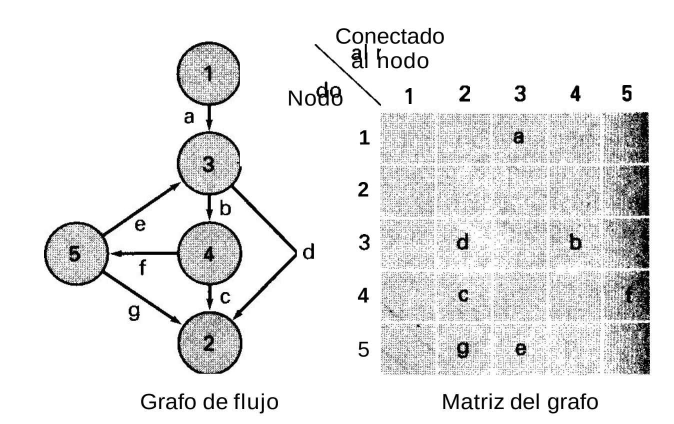


FIGURA 17.6. Matriz del grafo.

En la Figura 17.7, cada fila con dos o más entradas representa un nodo predicado. Por tanto, los cálculos aritméticos que se muestran a la derecha de la matriz de conexiones nos dan otro nuevo método de determinación de la complejidad ciclomática (Sección 17.4.2).

Beizer [BEI90] proporciona un tratamiento profundo de otros algoritmos matemáticos que se pueden aplicar a las matrices de grafos. Mediante estas técnicas, el análisis requerido para el diseño de casos de prueba se puede automatizar parcial o totalmente.

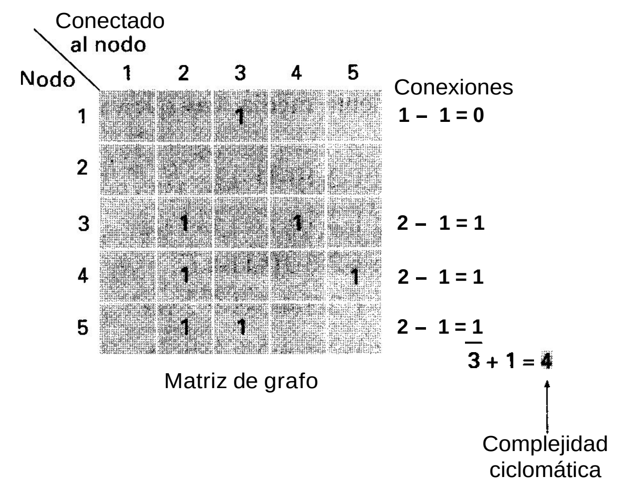


FIGURA 17.7. Matriz de conexiones.

17.5 PRUEBA DE LA ESTRUCTURA DE CONTROL

La técnica de prueba del camino básico descrita en la Sección 17.4 es una de las muchas técnicas para la *prueba de la estructura de control*. Aunque la prueba del camino básico es sencilla y altamente efectiva, no es suficiente por **sí** sola. En esta sección se tratan otras variantes de la prueba de estructura de control. Estas variantes amplían la cobertura de la prueba y mejoran la calidad de la prueba de caja blanca.

17.5.1. Prueba de condición[5]



CLAVE

Los errores son mucho **más** comunes en el entorno de los condiciones lógicas que en los sentencias de proceso secuencial.

La prueba de condición es un método de diseño de casos de prueba que ejercita las condiciones lógicas contenidas en el módulo de un programa. Una condición simple es una variable lógica o una expresión relacional, posiblemente precedida con un operador NOT («¬»). Una expresión relacional toma la siguiente forma

$$E_1 < operador\text{-}relacional > E_2$$

donde E_1 y E_2 son expresiones aritméticas y <*operador-relacional*> puede ser alguno de los siguientes: «<», «<=», «=», «≠» («=» desigualdad), «>», o «>=». Una *condición compuesta* está formada por dos o más condiciones simples, operadores lógicos y paréntesis. Suponemos que los operadores lógicos permitidos en una condición compuesta incluyen OR («|»), **AND** («&») y NOT («¬»). A una condición sin expresiones relacionales se la denomina *Expresión lógica*.

Por tanto, los tipos posibles de componentes en una condición pueden ser: un operador lógico, una variable lógica, un par de paréntesis lógicos (que rodean a una condición simple o compuesta), un operador relacional o una expresión aritmética.

Si una condición es incorrecta, entonces es incorrecto al menos un componente de la condición. Así, los tipos de errores de una condición pueden ser los siguientes:

- error en operador lógico (existencia de operadores lógicos incorrectos/desaparecidos/sobrantes)
- error en variable lógica
- error en paréntesis lógico
- error en operador relacional
- error en expresión aritmética.

El método de la *prueba de condiciones* se centra en la prueba de cada una de las condiciones del programa. Las estrategias de prueba de condiciones (tratadas posteriormente en este capítulo) tienen, generalmente, dos ventajas. La primera, que la cobertura de la prueba de una condición es sencilla. La segunda, que la cobertura de la prueba de las condiciones de un programa da una orientación para generar pruebas adicionales del programa.

El propósito de la prueba de condiciones es detectar, no sólo los errores en las condiciones de un programa, sino también otros errores en dicho programa. Si un conjunto de pruebas de un programa P es efectivo para detectar errores en las condiciones que se encuentran en P, es probable que el conjunto de pruebas también sea efectivo para detectar otros errores en el programa P. Además, si una estrategia de prueba es efectiva para detectar errores en una condición, entonces es probable que dicha estrategia también sea efectiva para detectar errores en el programa.

Se han propuesto una serie de estrategias de prueba de condiciones. La *prueba de ramificaciones* es, posiblemente, la estrategia de prueba de condiciones más sencilla. Para una condición compuesta C, es necesario ejecutar al menos una vez las ramas verdadera **y** falsa de C y cada condición simple de C [MYE79].



Cada vez que decides efectuar una prueba de condición, deberás evaluar cada condición en un intento por descubrir errores. ¡Este es un escondrijo ideal para los errores!

La *prueba del dominio* [WHI80] requiere la realización de tres o cuatro pruebas para una expresión relacional. Para una expresión relacional de la forma

$$E_1 < operador\text{-}relacional > E_2$$

se requieren tres pruebas, para comprobar que el valor de E_1 es mayor, igual o menor que el valor de E_2, respectivamente [HOW82]. Si el <*operador-relacional*> es incorrecto y E_1 y E_2 son correctos, entonces estas tres pruebas garantizan la detección de un error del operador relacional. Para detectar errores en E_1 y E_2, la prueba que haga el valor de E_1 mayor o menor que el de E_2 debe hacer que la diferencia entre estos dos valores sea lo más pequeña posible.

Para una expresión lógica con n variables, habrá que realizar las 2^n pruebas posibles $(n > 0)$. Esta estrategia puede detectar errores de un operador, de una variable y de un paréntesis lógico, pero sólo es práctica cuando el valor de n es pequeño.

[5] Las secciones 17.5.1. y 17.5.2. se han adaptado de [TAI89] con permiso del profesor K.C. Tai.

También se pueden obtener pruebas sensibles a error para expresiones lógicas [FOS84, TAI87]. Para una expresión lógica singular (una expresión lógica en la cual cada variable lógica sólo aparece una vez) con n variables lógicas ($n > 0$), podemos generar fácilmente un conjunto de pruebas con menos de 2" pruebas, de tal forma que ese grupo de pruebas garantice la detección de múltiples errores de operadores lógicos **y** también sea efectivo para detectar otros errores.

Tai [TAI89] sugiere una estrategia de prueba de condiciones que se basa en las técnicas destacadas anteriormente. La técnica, denominada BRO* *(prueba del operador relacional y de ramificación)*, garantiza la detección de errores de operadores relacionales **y** de ramificaciones en una condición dada, en la que todas las variables lógicas **y** operadores relacionales de la condición aparecen sólo una vez **y** no tienen variables en común.

La estrategia BRO utiliza restricciones de condición para una condición C. Una restricción de condición para C con n condiciones simples se define como (D_1, D_2,...,D_n), donde D_i ($0 < i \leq n$) es un símbolo que especifica una restricción sobre el resultado de la i-ésima condición simple de la condición C. Una restricción de condición D para una condición C se cubre o se trata en una ejecución de C, si durante esta ejecución de C, el resultado de cada condición simple de C satisface la restricción correspondiente de D.

Para una variable lógica B, especificamos una restricción sobre el resultado de B, que consiste en que B tiene que ser verdadero (v) o falso (f). De forma similar, para una expresión relacional, se utilizan los símbolos >, = **y** < para especificar restricciones sobre el resultado de la expresión.

Como ejemplo, consideremos la condición

$$C_1: \quad B_1 \,\&\, B_2$$

donde B, **y** B, son variables lógicas. La restricción de condición para C_1 es de la forma (D_1, D_2), donde D, y D, son «v» o «f». El valor (v, f) es una restricción de condición para C_1 **y** se cubre mediante la prueba que hace que el valor de B, sea verdadero **y** el valor de B, sea falso. La estrategia de prueba BRO requiere que el conjunto de restricciones {(v, v), (f, v), (v, f)} sea cubierto mediante las ejecuciones de C,. Si C, es incorrecta por uno o más errores de operador lógico, por lo menos un par del conjunto de restricciones forzará el fallo de C,.

Como segundo ejemplo, consideremos una condición de la forma

$$C_2: \quad B_1 \,\&\, (E_3 = E_4)$$

donde B_1 es una expresión lógica y E, **y** E, son expresiones aritméticas. Una restricción de condición para C, es de la forma (D_1, D_2), donde D_1 es «v» o «f» **y** D, es >, = o <. Puesto que C_2 es igual que C_1, excepto en que la segunda condición simple de C_2 es una expresión relacional, podemos construir un conjunto de restricciones para C_2 mediante la modificación del conjunto de restricciones {(v, v), (f, v), (v, f)} definido para C,. Obsérvese que «v» para $(E, = E_4)$ implica «=» **y** que «f» para $(E_3 = E_4)$ implica «< »o «>». Al reemplazar (v, v) **y** (f, v) por (v, =) **y** (f, =), respectivamente **y** reemplazando (v, f) por (v, <) **y** (v,>), el conjunto de restricciones resultante para C, es {(v, =), (f, =), **(v,** <), (v, >)}.La cobertura del conjunto de restricciones anterior garantizará la detección de errores del operador relacional o lógico en C_2.

Como tercer ejemplo, consideremos una condición de la forma

$$C_3: \quad (E_1 > E_2) \,\&\, (E_3 \doteq E_4)$$

donde E,, E_2, E, y E, son expresiones aritméticas. Una restricción de condición para C, es de la forma (D_1, D_2), donde todos los D, **y** D, son >, = o <. Puesto **que** C_3 es igual que C, excepto en que la primera condición simple de C, es una expresión relacional, podemos construir un conjunto de restricciones para C_3 mediante la modificación del conjunto de restricciones para C,, obteniendo

{(>, =), (=, =), (<, =), (>, >), (>. <)}

La cobertura de este conjunto de restricciones **garan**tizará la detección de errores de operador relacional en C,.

17.5.2. Prueba del flujo de datos

El método de *prueba de flujo* ***de*** *datos* selecciona caminos de prueba de un programa de acuerdo con la ubicación de las definiciones **y** los usos de las variables del programa. Se han estudiado **y** comparado varias estrategias de prueba de flujo de datos (por ejemplo, [FRA88], [NTA88], [FRA93]).

Para ilustrar el enfoque de prueba de flujo de datos, supongamos que a cada sentencia de un programa se le asigna un número único de sentencia **y** que las funciones no modifican sus parámetros o las variables globales. Para una sentencia con S como número de sentencia,

DEF(S) = **{** X I la sentencia **S** contiene una definición de **X }**

USO(S) = { X I la sentencia **S** contiene un uso de X}

Si la sentencia S es una sentencia *if* o de bucle, su conjunto DEF estará vacío y su conjunto USE estará basado en la condición de la sentencia *S*. La definición de una variable *X* en una sentencia *S* se dice que está *viva* en una sentencia S' si existe un camino de la sentencia *S* a la sentencia S' que no contenga otra definición de *X*.



Es para realista asumir que la prueba del flujo de datas puede ser usada ampliamente cuando probamos grandes sistemas. En cualquier rasa, puede ser utilizada en areas del software que sean sospechosos.

* En ingles, Branch arid Relational Operator

Una ***cadena de definición-uso (o*** cadena DU) de una variable X tiene la forma $[X, S, S']$, donde **S** y **S'** son números de sentencia, X está en DEF(S) y en USO(S') y la definición de X en la sentencia **S** está viva en la sentencia **S'.**

Una sencilla estrategia de prueba de flujo de datos se basa en requerir que se cubra al menos una vez cada cadena DU. Esta estrategia se conoce como ***estrategia de prueba*** DU. Se ha demostrado que la prueba DU no garantiza que se cubran todas las ramificaciones de un programa. Sin embargo, solamente no se garantiza el cubrimiento de una ramificación en situaciones raras como las construcciones ***if-then-else*** en las que la parte ***then*** no tiene ninguna definición de variable y no existe la parte ***else.*** En esta situación, la prueba DU no cubre necesariamente la rama ***else*** de la sentencia $superior.

Las estrategias de prueba de flujo de datos son Útiles para seleccionar caminos de prueba de un programa que contenga sentencias ***if o de bucles*** anidados. Para ilustrar esto, consideremos la aplicación de la prueba DU para seleccionar caminos de prueba para el LDP que sigue:

```
proc x
    B1:
    do while C1
       if C2
          then
            if C4
               then B4;
               else B5;
            endif;
          else
            if C3
               then b2;
               else B3;
          endif;
       endif;
     enddo;
      B6;
  end proc;
```

Para aplicar la estrategia de prueba DU para seleccionar caminos de prueba del diagrama de flujo de control, necesitamos conocer las definiciones y los usos de las variables de cada condición o cada bloque del LDP. Asumimos que la variable X es definida en la Última sentencia de los bloques B1, B2, B3, B4 y B5, y que se usa en la primera sentencia de los bloques B2, B3, B4, B5 **y B6.** La estrategia de prueba DU requiere una ejecución del camino más corto de cada B_i, $0 < i \leq 5$, a cada B_j, $1 < j \leq 6$. (Tal prueba también cubre cualquier uso de la variable X en las condiciones C1, C2, C3 y C4). Aunque hay veinticinco cadenas DU de la variable X, sólo necesitamos cinco caminos para cubrir la cadena DU. La razón es que se necesitan cinco caminos para cubrir la cadena DU de X desde B_i, $0 < i \leq 5$, hasta B6, y las otras cadenas DU se pueden cubrir haciendo que esos cinco caminos contengan iteraciones del bucle.

Dado que las sentencias de un programa están relacionadas entre **sí** de acuerdo con las definiciones de las variables, el enfoque de prueba de flujo de datos es efectivo para la protección contra errores. Sin embargo, los problemas de medida de la cobertura de la prueba y de selección de caminos de prueba para la prueba de flujo de datos son más difíciles que los correspondientes problemas para la prueba de condiciones.

17.5.3. Prueba de bucles

Los bucles son la piedra angular de la inmensa mayoría de los algoritmos implementados en software. **Y** sin embargo, les prestamos normalmente poca atención cuando llevamos a cabo las pruebas del software.



Las estructuras de bucles complejos es otro lugar propenso a errores. Por tonto, es muy valioso realizar diseños de pruebas que ejerciten completamente los estructuras bucle.

La prueba de bucles es una técnica de prueba de caja blanca que se centra exclusivamente en la validez de las construcciones de bucles. Se pueden definir cuatro clases diferentes de bucles **[BEI90]**: ***bucles simples, bucles concatenados, bucles anidados*** y ***bucles no estructurados*** (Fig. 17.8).

Bucles simples. A los bucles simples se les debe aplicar el siguiente conjunto de pruebas, donde ***n*** es el número máximo de pasos permitidos por el bucle:

1. pasar por alto totalmente el bucle
2. pasar una sola vez por el bucle
3. pasar dos veces por el bucle
4. hacer ***m*** pasos por el bucle con ***m < n***
5. hacer ***n*** – 1, ***n*** y ***n+l*** pasos por el bucle

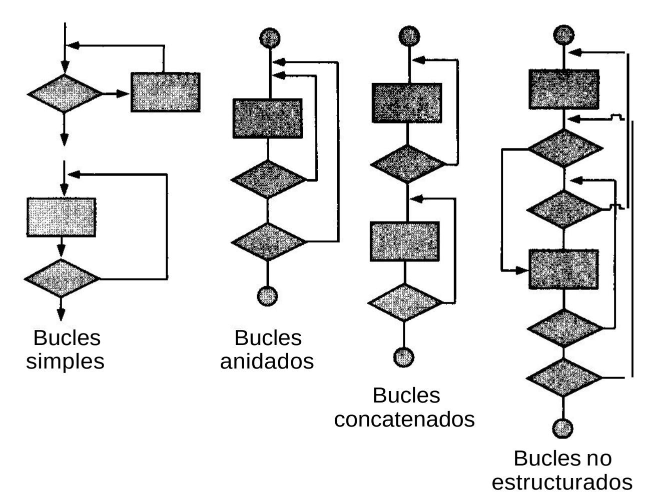


FIGURA 17.8. Clases de bucles.

Bucles anidados. Si extendiéramos el enfoque de prueba de los bucles simples a los bucles anidados, el número de posibles pruebas aumentaría geométricamente a medida que aumenta el nivel de anidamiento. Esto llevaría a un número impracticable de pruebas. Beizer [BEI90] sugiere un enfoque que ayuda a reducir el número de pruebas:

1. Comenzar por el bucle más interior. Establecer o configurar los demás bucles con sus valores mínimos.
2. Llevar a cabo las pruebas de bucles simples para el bucle más interior, mientras se mantienen los parámetros de iteración (por ejemplo, contador del bucle) de los bucles externos en sus valores mínimos. Añadir otras pruebas para valores fuera de rango o excluidos.
3. Progresar hacia fuera, llevando a cabo pruebas para el siguiente bucle, pero manteniendo todos los bucles externos en sus valores mínimos y los demás bucles anidados en sus valores «típicos».
4. Continuar hasta que se hayan probado todos los bucles.

Bucles concatenados. Los bucles concatenados se pueden probar mediante el enfoque anteriormente definido para los bucles simples, mientras cada uno de los bucles sea independiente del resto. Sin embargo, si hay dos bucles concatenados y se usa el controlador del bucle 1 como valor inicial del bucle *2*, entonces los bucles no son independientes. Cuando los bucles no son independientes, se recomienda usar el enfoque aplicado para los bucles anidados.



No debes probar los bucles no estructurados. Rediséñalos.

Bucles no estructurados. Siempre que sea posible, esta clase de bucles se deben *rediseñar* para que se ajusten a las construcciones de programación estructurada (Capítulo 16).

17.6 PRUEBA DE CAJA NEGRA

Las pruebas de caja negra, también denominada *prueba de comportamiento,* se centran en los requisitos funcionales del software. O sea, la prueba de caja negra permite al ingeniero del software obtener conjuntos de condiciones de entrada que ejerciten completamente todos los requisitos funcionales de un programa. La prueba de caja negra no es una alternativa a las técnicas de prueba de caja blanca. **Más** bien se trata de un enfoque complementario que intenta descubrir diferentes tipos de errores que los métodos de caja blanca.

La prueba de caja negra intenta encontrar errores de las siguientes categorías: (1) funciones incorrectas o ausentes, (2) errores de interfaz, (3) errores en estructuras de datos o en accesos a bases de datos externas, (4) errores de rendimiento y (5) errores de inicialización y de terminación.

A diferencia de la prueba de caja blanca, que se lleva a cabo previamente en el proceso de prueba, la prueba de caja negra tiende a aplicarse durante fases posteriores de la prueba (véase el Capítulo 18). Ya que la prueba de caja negra ignora intencionadamente la estructura de control, centra su atención en el campo de la información. Las pruebas se diseñan para responder a las siguientes preguntas:

- ¿Cómo se prueba la validez funcional?
- ¿Cómo se prueba el rendimiento y el comportamiento del sistema?
- ¿Qué clases de entrada compondrán unos buenos casos de prueba?
- ¿Es el sistema particularmente sensible a ciertos valores de entrada?
- ¿De qué forma están aislados los límites de una clase de datos?
- ¿Qué volúmenes y niveles de datos tolerará el sistema?
- ¿Qué efectos sobre la operación del sistema tendrán combinaciones específicas de datos?

Mediante las técnicas de prueba de caja negra se obtiene un conjunto de casos de prueba que satisfacen los siguientes criterios [MYE79]: (1) casos de prueba que reducen, en un coeficiente que es mayor que uno, el número de casos de prueba adicionales que se deben diseñar para alcanzar una prueba razonable y (2) casos de prueba que nos dicen algo sobre la presencia o ausencia de clases de errores en lugar de errores asociados solamente con la prueba que estamos realizando.

17.6.1. Métodos de prueba basados en grafos

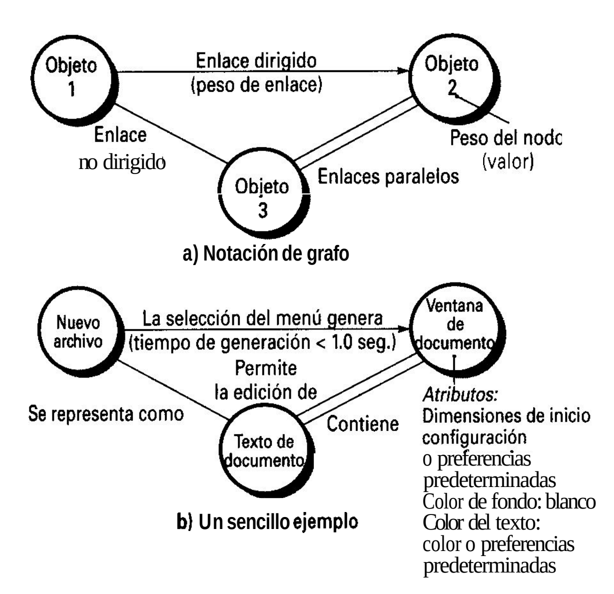
El primer paso en la prueba de caja negra es entender los objetos[6] que se modelan en el software y las relaciones que conectan a estos objetos. Una vez que se ha llevado a cabo esto, el siguiente paso es definir una serie de pruebas que verifiquen que «todos los objetos tienen entre ellos las relaciones esperadas» [BEI95]. Dicho de otra manera, la prueba del software empieza creando un grafo de objetos importantes y sus relaciones, y después

[6] En este contexto, el término «objeto» comprende los objetos de datos que se estudiaron en los Capítulos 11 y 12 así como objetos de programa tales como módulos o colecciones de sentencias del lenguaje de programación.

diseñando una serie de pruebas que cubran el grafo de manera que se ejerciten todos los objetos y sus relaciones para descubrir los errores.



Un grafo representa la relación entre objeto dato y objeto programa, permitiéndonos derivar casos de prueba que buscan errores asociados con estas relaciones.



RGURA 17.9.

Para llevar a cabo estos pasos, el ingeniero del software empieza creando un *grafo* —una colección de *nodos* que representan objetos; *enlaces* que representan las relaciones entre **los** objetos; *pesos de nodos* que describen las propiedades de un nodo (por ejemplo, un valor específico de datos o comportamiento de estado) y *pesos de enlaces* que describen alguna característica de un enlace—[7].

En la Figura 17.9a se muestra una representación simbólica de un grafo. Los nodos se representan como círculos conectados por enlaces que toman diferentes formas. Un *enlace dirigido* (representado por una flecha) indica que una relación se mueve sólo en una dirección. Un *enlace bidireccional*, también denominado *enlace simétrico*, implica que la relación se aplica en ambos sentidos. Los enlaces paralelos se usan cuando se establecen diferentes relaciones entre los nodos del grafo.

Como ejemplo sencillo, consideremos una parte de un grafo de una aplicación de un procesador de texto (Fig. 17.9b) donde:

objeto n.º 1 = **selección en el menú de archivo nuevo**

objeto n.º 2 = **ventana del documento**

objeto n.º 3 = **texto del documento**

Como se muestra en la figura, una selección del menú en **archivo nuevo** genera una **ventana del documento.** El peso del nodo de **ventana del documento** proporciona una lista de los atributos de la ventana que se esperan cuando se genera una ventana. El peso del enlace indica que la ventana se tiene que generar en menos de **1.0** segundos. Un enlace no dirigido establece una relación simétrica entre **selección en el menú de archivo nuevo** y **texto del documento,** y los enlaces paralelos indican las relaciones entre la **ventana del documento** y el **texto del documento.** En realidad, se debería generar un grafo bastante más detallado como precursor al diseño de casos de prueba. El ingeniero del software obtiene entonces casos de prueba atravesando el grafo y cubriendo cada una de las relaciones mostradas. Estos casos de prueba están diseñados para intentar encontrar errores en alguna de las relaciones.

Beizer [BEI95] describe un número de métodos de prueba de comportamiento que pueden hacer uso de los grafos:

Modelado del flujo de transacción. Los nodos representan los pasos de alguna transacción (por ejemplo, los pasos necesarios para una reserva en una línea aérea usando un servicio en línea), y los enlaces representan las conexiones lógicas entre los pasos (por ejemplo, *vuelo.información.entrada* es seguida de *validación/disponibilidad.procesamiento*). El diagrama de flujo de datos (Capítulo 12) puede usarse para ayudar en la creación de grafos de este tipo.

Modelado de estado finito. Los nodos representan diferentes estados del software observables por el usuario (por ejemplo, cada una de las «pantallas» que aparecen cuando un telefonista coge una petición por teléfono), y los enlaces representan las transiciones que ocurren para moverse de estado a estado (por ejemplo, *petición-información se verifica durante inventario-disponibilidad-búsqueda* y es seguido por *cliente-factura-información-entrada*). El diagrama estado-transición (Capítulo 12) puede usarse para ayudar en la creación de grafos de este tipo.

Modelado del flujo de datos. Los nodos son objetos de datos y los enlaces son las transformaciones que ocurren para convertir un objeto de datos en otro. Por ejemplo, el nodo **FICA.impuesto.retenido** (FIR) se calcula de **brutos.sueldos** (BS) usando la relación FIR = 0,62 **X** BS.

Modelado de planificación. Los nodos son objetos de programa y los enlaces son las conexiones secuenciales entre esos objetos. Los pesos de enlace se usan para especificar los tiempos de ejecución requeridos al ejecutarse el programa.

[7] Si los conceptos anteriores suenan vagamente familiares, recordemos que los grafos se usaron también en la Sección 17.4.1 para crear un grafo del programa para el método de la prueba del camino básico. Los nodos del grafo del programa contenían instrucciones (objetos de programa) caracterizados como representaciones de diseño procedimental o como código fuente y los enlaces dirigidos indicaban el flujo de control entre estos objetos del programa. **Aquí** se extiende el uso de los grafos para incluir la prueba de caja negra.

Un estudio detallado de cada uno de estos métodos de prueba basados en grafos está más allá del alcance de este libro. El lector interesado debería consultar [BEI95] para ver un estudio detallado. Merece la pena, sin embargo, proporcionar un resumen genérico del enfoque de pruebas basadas en grafos.

¿Cuáles son las actividades generales requeridas durante la prueba basada en un grafo?

Las pruebas basadas en grafos empiezan con la definición de todos los nodos y pesos de nodos. O sea, se identifican los objetos y los atributos. El modelo de datos (Capítulo **12)** puede usarse como punto de partida, pero es importante tener en cuenta que muchos nodos pueden ser objetos de programa (no representados explícitamente en el modelo de datos). Para proporcionar una indicación de los puntos de inicio y final del grafo, es útil definir unos nodos de entrada y salida.

Una vez que se han identificado los nodos, se deberían establecer los enlaces y los pesos de enlaces. En general, conviene nombrar los enlaces, aunque los enlaces que representan el flujo de control entre los objetos de programa no es necesario nombrarlos.

En muchos casos, el modelo de grafo puede tener bucles (por ejemplo, un camino a través del grafo en el que se encuentran uno o más nodos más de una vez). La prueba de bucle (Sección 17.5.3) se puede aplicar también a nivel de comportamiento (de caja negra). El grafo ayudará a identificar aquellos bucles que hay que probar.

Cada relación es estudiada separadamente, de manera que se puedan obtener casos de prueba. La *transitividad* de relaciones secuenciales es estudiada para determinar cómo se propaga el impacto de las relaciones a través de objetos definidos en el grafo. La transitividad puede ilustrarse considerando tres objetos **X, Y** y Z. Consideremos las siguientes relaciones:

X *es necesaria para calcular* **Y**

Y *es necesaria para calcular* **Z**

Por tanto, se ha establecido una relación transitiva entre **X** y **Z:**

X *es necesaria para calcular* **Z**

Basándose en esta relación transitiva, las pruebas para encontrar errores en el cálculo de **Z** deben considerar una variedad de valores para **X** e **Y.**

La *simetría* de una relación (enlace de grafo) es también una importante directriz para diseñar casos de prueba. Si un enlace es bidireccional (simétrico), es importante probar esta característica. La característica *UNDO* [BEI95] (deshacer) en muchas aplicaciones para ordenadores personales implementa una limitada simetría. Es decir, *UNDO* permite deshacer una acción después de haberse completado. Esto debería probarse minuciosamente y todas las excepciones (por ejemplo, lugares donde no se puede usar *UNDO)* deberían apuntarse. Finalmente, todos los nodos del grafo deberían tener una relación que los lleve de vuelta a ellos mismos; en esencia, un bucle de «no acción» o «acción nula». Estas relaciones *reflexivas* deberían probarse también.

Cuando empieza el diseño de casos de prueba, el primer objetivo es conseguir la *cobertura de nodo.* Esto significa que las pruebas deberían diseñarse para demostrar que ningún nodo se ha omitido inadvertidamente y que los pesos de nodos (atributos de objetos) son correctos.

A continuación, se trata la *cobertura de enlaces.* Todas las relaciones se prueban basándose en **sus** propiedades. Por ejemplo, una relación simétrica se prueba para demostrar que es, de hecho, bidireccional. Una relación transitiva se prueba para demostrar que existe transitividad. Una relación reflexiva se prueba para asegurarse de que hay un bucle nulo presente. Cuando se han especificado los pesos de enlace, se diseñan las pruebas para demostrar que estos pesos son válidos. Finalmente, se invocan las pruebas de bucles (Sección 17.5.3).

17.6.2. Partición equivalente

La *partición equivalente* es un método de prueba de caja negra que divide el campo de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba. Un caso de prueba ideal descubre de forma inmediata una clase de errores (por ejemplo, proceso incorrecto de todos los datos de carácter) que, de otro modo, requerirían la ejecución de muchos casos antes de detectar el error genérico. La partición equivalente se dirige a la definición de casos de prueba que descubran clases de errores, reduciendo así el número total de casos de prueba que hay que desarrollar.



las clases de entrada san conocidas relativamente temprano en el procesa de software. Por esto razón, comenzamos pensando en la partición equivalente una vez el diseña ha sido creada.

El diseño de casos de prueba para la partición equivalente se basa en una evaluación de las clases de equivalencia para una condición de entrada. Mediante conceptos introducidos en la sección anterior, si un conjunto de objetos puede unirse por medio de relaciones simétricas, transitivas y reflexivas, entonces existe una clase de equivalencia [BEI95]. Una *clase de equivalencia* representa un conjunto de estados válidos o no válidos para condiciones de entrada. Típicamente, una condición de entrada es un valor numérico específico, un rango de valores, un conjunto de valores relacionados o una condición lógica. Las clases de equivalencia se pueden definir de acuerdo con las siguientes directrices:

1. Si una condición de entrada especifica un *rango,* se define una clase de equivalencia válida y dos no válidas.

2. Si una condición de entrada requiere un valor específico, se define una clase de equivalencia válida y dos no válidas.
3. Si una condición de entrada especifica un miembro de un conjunto, se define una clase de equivalencia válida y una no válida.
4. Si una condición de entrada es lógica, se define una clase de equivalencia válida y una no válida.

Como ejemplo, consideremos los datos contenidos en una aplicación de automatización bancaria. El usuario puede «llamar» al banco usando su ordenador personal, dar su contraseña de seis dígitos y continuar con una serie de órdenes tecleadas que desencadenarán varias funciones bancarias. El software proporcionado por la aplicación bancaria acepta datos de la siguiente forma:

Código de área: en blanco o un número de tres dígitos
Prefijo: número de tres dígitos que no comience por 0 o 1
Sufijo: número de cuatro dígitos
Contraseña: valor alfanumérico de seis dígitos
Órdenes: «comprobar», «depositar», «pagar factura», etc.

Las condiciones de entrada asociadas con cada elemento de la aplicación bancaria se pueden especificar como:

Código de área:	condición de entrada, *lógica*—el código de área puede estar o no presente condición de entrada, *rango*—valores definidos entre 200 y 999, con excepciones específicas
Prefijo:	condición de entrada, rango —valor especificado > 200 sin dígitos 0
Sufijo:	condición de entrada, valor —longitud de cuatro dígitos
Contraseña:	condición de entrada, lógica —la palabra clave puede estar o no presente; condición de entrada, valor —cadena de seis caracteres
Orden:	condición de entrada, *conjunto*—contenida en las órdenes listadas anteriormente

Aplicando las directrices para la obtención de clases de equivalencia, se pueden desarrollar y ejecutar casos de prueba para cada elemento de datos del campo de entrada. Los casos de prueba se seleccionan de forma que se ejercite el mayor número de atributos de cada clase de equivalencia a la vez.

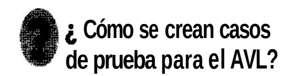
17.6.3. Análisis de valores límite

Por razones que no están del todo claras, los errores tienden a darse más en los límites del campo de entrada que en el «centro». Por ello, se ha desarrollado el *análisis* de valores límites (AVL) como técnica de prueba. El análisis de valores límite nos lleva a una elección de casos de prueba que ejerciten **los** valores límite.



AVL amplía la partición equivalente para fijarse sobre datos en el «límite» de una clase de equivalencia.

El análisis de valores límite es una técnica de diseño de casos de prueba que complementa a la partición equivalente. En lugar de seleccionar cualquier elemento de una clase de equivalencia, el **AVL** lleva a la elección de casos de prueba en los «extremos» de la clase. En lugar de centrarse solamente en las condiciones de entrada, el AVL obtiene casos de prueba también para el campo de salida [MYE79].



Las directrices de AVL son similares en muchos aspectos a las que proporciona la partición equivalente:

1. Si una condición de entrada especifica un rango delimitado por los valores a y ***b*,** se deben diseñar casos de prueba para los valores a y *b*, y para los valores justo por debajo y justo por encima de a y *b*, respectivamente.
2. Si una condición de entrada especifica un número de valores, se deben desarrollar casos de prueba que ejerciten los valores máximo y mínimo. También se deben probar los valores justo por encima y justo por debajo del máximo y del mínimo.
3. Aplicar las directrices 1 y 2 a las condiciones de salida. Por ejemplo, supongamos que se requiere una tabla de «temperatura / presión» como salida de un programa de análisis de ingeniería. Se deben diseñar casos de prueba que creen un informe de salida que produzca el máximo (y el mínimo) número permitido de entradas en la tabla.
4. Si las estructuras de datos internas tienen límites preestablecidos (por ejemplo, una matriz que tenga un límite definido de 100 entradas), hay que asegurarse de diseñar un caso de prueba que ejercite la estructura de datos en sus límites.

La mayoría de los ingenieros del software llevan a cabo de forma intuitiva alguna forma de AVL. Aplicando las directrices que se acaban de exponer, la prueba de límites será más completa y, por tanto, tendrá una mayor probabilidad de detectar errores.

17.6.4. Prueba de comparación

Hay situaciones (por ejemplo, aviónica de aeronaves, control de planta nuclear) en las que la fiabilidad del software es algo absolutamente crítico. En ese tipo de

aplicaciones, a menudo se utiliza hardware y software redundante para minimizar la posibilidad de error. Cuando se desarrolla software redundante, varios equipos de ingeniería del software separados desarrollan versiones independientes de una aplicación, usando las mismas especificaciones. En esas situaciones, se deben probar todas las versiones con los mismos datos de prueba, para asegurar que todas proporcionan una salida idéntica. Luego, se ejecutan todas las versiones en paralelo y se hace una comparación en tiempo real de los resultados, para garantizar la consistencia.

Con las lecciones aprendidas de los sistemas redundantes, los investigadores (por ejemplo, [BRI87]) han sugerido que, para las aplicaciones críticas, se deben desarrollar versiones de software independientes, incluso aunque sólo se vaya a distribuir una de las versiones en el sistema final basado en computadora. Esas versiones independientes son la base de una técnica de prueba de caja negra denominada *prueba de comparación* o *prueba mano a mano* [KNI89].

Cuando se han producido múltiples implementaciones de la misma especificación, a cada versión del software se le proporciona como entrada los casos de prueba diseñados mediante alguna otra técnica de caja negra (por ejemplo, la partición equivalente). Si las salidas producidas por las distintas versiones son idénticas, se asume que todas las implementaciones son correctas. Si la salida es diferente, se investigan todas las aplicaciones para determinar el defecto responsable de la diferencia en una o más versiones. En la mayoría de los casos, la comparación de las salidas se puede llevar a cabo mediante una herramienta automática.

17.6.5. Prueba de la tabla ortogonal

Hay muchas aplicaciones en que el dominio de entrada es relativamente limitado. Es decir, el número de parámetros de entrada es pequeño y los valores de cada uno de los parámetros está claramente delimitado. Cuando estos números son muy pequeños (por ejemplo, 3 parámetros de entrada tomando 3 valores diferentes), es posible considerar cada permutación de entrada y comprobar exhaustivamente el proceso del dominio de entrada. En cualquier caso, cuando el número de valores de entrada crece y el número de valores diferentes para cada elemento de dato se incrementa, la prueba exhaustiva se hace impracticable o imposible.



La prueba de la tabla ortogonal permite diseñar casos de prueba que facilitan una cobertura máxima de prueba con un número razonable de casos de prueba.

La prueba de la tabla ortogonal puede aplicarse a problemas en que el dominio de entrada es relativamente pequeño pero demasiado grande para posibilitar pruebas exhaustivas. El método de prueba de la tabla ortogonal es particularmente útil al encontrar errores asociados con fallos localizados —una categoría de error asociada con defectos de la lógica dentro de un componente software—.

Para ilustrar la diferencia entre la prueba de la tabla ortogonal y una aproximación más convencional «un elemento de entrada distinto cada vez», considerar un sistema que tiene tres elementos de entrada, **X,** Y y Z. Cada uno de estos elementos de entrada tiene tres valores diferentes. Hay $3^3 = 27$ posibles casos de prueba. Phadke [PHA97] sugiere una visión geométrica de los posibles casos de prueba asociados con X, Y y Z, según se ilustra en la Figura 17.10. Observando la figura, cada elemento de entrada en un momento dado puede modificarse secuencialmente en cada eje de entrada.

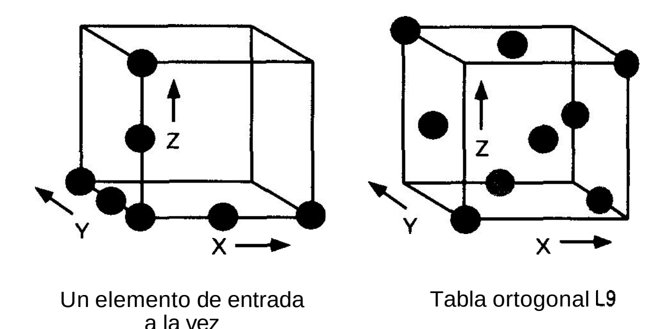


FIGURA 17.10. Una vista geométrica de los casos.

Esto da como resultado un alcance relativamente limitado al dominio de entrada (representado en la figura por el cubo de la izquierda).

Cuando se realiza la prueba de la tabla ortogonal, se crea una *tabla ortogonal L9* de casos de prueba. La tabla ortogonal L9 tiene una «propiedad de equilibrio)) [PHA97]. Es decir, los casos de prueba (representados por puntos negros en la figura) están «uniformemente dispersos en el dominio de prueba», según se ilustra en el cubo de la derecha de la Figura 17.10. El alcance de la prueba por todo el dominio de entrada es más completo.

Para ilustrar el uso de la tabla ortogonal L9, considerar la función *enviar* para una aplicación de un fax. Cuatro parámetros, P1, P2, P3 y P4 se pasan a la función *enviar*. Cada uno tiene tres valores diferentes. Por ejemplo, P1 toma los valores:

P1 = 1, enviar ahora
P2 = 2, enviar dentro de una hora
P3 = 3, enviar después de media noche

P2, P3, y P4 podrán tomar también los valores 1, 2 y 3, representando otras funciones *enviar*.

Si se elige la estrategia de prueba «un elemento de entrada distinto cada vez», se especifica la siguiente secuencia de pruebas (P1,P2,P3,P4): (1,1,1,1), (2,1,1,1),

(3,1,1,1), (1,2,1,1), (1,3,1,1), (1,1,2,1), (1,1,3,1), (1,1,1,2), y (1,1,1,3). Phadke [PHA97] valoraestos casos de prueba de la siguiente manera:

> Cada uno de estos casos de prueba son Útiles, Únicamente cuando estos parámetros de prueba no se influyen mutuamente. Pueden detectar fallos lógicos cuando el valor de un parámetro produce un mal funcionamiento del software. Estos fallos pueden llamarse fallos de modalidad simple. El método no puede detectar fallos lógicos que causen un mal funcionamiento cuando dos o más parámetros simultáneamente toman determinados valores; es decir, no se pueden detectar interacciones. **Así,** esta habilidad para detectar fallos es limitada.

Dados un número relativamente pequeño de parámetros de entrada y valores diferentes, es posible realizar una prueba exhaustiva. El número de pruebas requeridas es 3–81 —grande pero manejable—. Todos los fallos asociados con la permutación de los datos serán encontrados, pero el esfuerzo requerido es relativamente alto.

[illegible]	Parámetros de prueba			
	P_1	P_2	P_3	P_4
	1	1	1	1
	1	2	2	2
	1	3	3	3
	2	1	2	3
	2	2	3	1
	2	3	1	2
	3	1	3	2
	3	2	1	3
	3	3	2	1

FIGURA 17.11. Una tabla ortogonal L9.

La prueba de la tabla ortogonal nos permite proporcionar una buena cobertura de prueba con bastantes menos casos de prueba que en la estrategia exhaustiva. Una tabla ortogonal L9 para la función de envío del fax se describe en la Figura 17.11.

Phadke [PHA97] valora el resultado de las pruebas utilizando la tabla ortogonal de la siguiente manera:

> **Detecta y aísla todos los fallos de modalidad simple.** Un fallo de modalidad simple es un problema que afecta a un solo parámetro. Por ejemplo, si todos los casos de prueba del factor P1 = 1 causan una condición de error, nos encontramos con el fallo de modalidad simple. En los ejemplos de prueba 1, 2 y *3* [Fig. 17.111 se encontrarán errores. Analizando la información en que las pruebas muestran errores, se puede identificar que valores del parámetro causan el error. En este ejemplo, anotamos que las pruebas 1, 2 y *3* causan un error, lo que permite aislar [el proceso lógico asociado con «enviar ahora» (Pl = 1)] la fuente del error. **El** aislamiento del fallo es importante para solucionar el error.
>
> **Detecta todos los fallos de modalidad doble. Si** existe un problema donde están afectados dos parámetros que intervienen conjuntamente, se llama fallo de modalidad doble. En efecto, un fallo de modalidad doble es una indicación de incompatibilidad o de imposibilidad de interacción entre dos parámetros.
>
> **Fallos multimodales.** Las tablas ortogonales [del tipo indicado] pueden asegurar la detección Únicamente de fallos de modalidad simple o doble. Sin embargo, muchos fallos en modalidad múltiple pueden ser detectados a través de estas pruebas.

Se puede encontrar un estudio detallado sobre la prueba de tabla ortogonal en [PHA89].

17.7 PRUEBA DE ENTORNOS ESPECIALIZADOS, ARQUITECTURAS Y APLICACIONES

A medida que el software de computadora se ha hecho más complejo, ha crecido también la necesidad de enfoques de pruebas especializados. Los métodos de prueba de caja blanca y de caja negra tratados en las Secciones 17.5 y 17.6 son aplicables a todos los entornos, arquitecturas y aplicaciones pero a veces se necesitan unas directrices y enfoques Únicos para las pruebas. En esta sección se estudian directrices de prueba para entornos, arquitecturas y aplicaciones especializadas que pueden encontrarse los ingenieros del software.

17.7.1. Prueba de interfaces gráficas de usuario (IGUs)

Las interfaces gráficas de usuario (IGUs) presentan interesantes desafíos para los ingenieros del software. Debido a los componentes reutilizables provistos como parte de los entornos de desarrollo de las GUI, la creación de la interfaz de usuario se ha convertido en menos costosa en tiempo y más exacta. Al mismo tiempo, la complejidad de las IGUs ha aumentado, originando más dificultad en el diseño y ejecución de los casos de prueba.

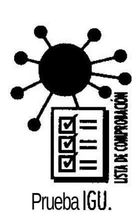
Referencia cruzada

Una guía para el diseño de IGU se presenta en el Capítulo 15.

Dado que las IGUs modernas tienen la misma apariencia y filosofía, se pueden obtener una serie de pruebas estándar. Los grafos de modelado de estado finito (Sección 16.6.1) pueden ser utilizados para realizar pruebas que vayan dirigidas sobre datos específicos y programas objeto que sean relevantes para las IGUs.

Considerando el gran número de permutaciones asociadas con las operaciones IGU, sería necesario para

probar el utilizar herramientas automáticas. Una amplía lista de herramientas de prueba de IGU han aparecido en el mercado en los Últimos años. Para profundizar en el tema, ver el Capítulo 31.



Prueba IGU.

17.7.2. Prueba de arquitectura cliente/servidor

La arquitectura cliente/servidor (C/S) representa un desafío significativo para los responsables de la pruebas del software. La naturaleza distribuida de los entornos cliente/servidor, los aspectos de rendimiento asociados con el proceso de transacciones, la presencia potencial de diferentes plataformas hardware, las complejidades de las comunicaciones de red, la necesidad de servir a múltiples clientes desde una base de datos centralizada (o en algunos casos, distribuida) y los requisitos de coordinación impuestos al servidor se combinan todos para hacer las pruebas de la arquitectura C/S y el software residente en ellas, considerablemente más difíciles que la prueba de aplicaciones individuales. De hecho, estudios recientes sobre la industria indican un significativo aumento en el tiempo invertido y los costes de las pruebas cuando se desarrollan entornos C/S.

Referencia cruzada

Lo ingenierío del software **cliente/servidor** se presento en el Capítulo **28.**

17.7.3. Prueba de la documentación y facilidades de ayuda

El término apruebas del software» hace imaginarnos gran cantidad de casos de prueba preparados para ejecutar programas de computadora y los datos que manejan. Recordando la definición de software presentada en el primer capítulo de este libro, es importante darse cuenta de que la prueba debe extenderse al tercer elemento de la configuración del software —la documentación—.

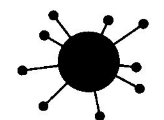
Los errores en la documentación pueden ser tan destructivos para la aceptación del programa, como los errores en los datos o en el código fuente. Nada es más frustrante que seguir fielmente el manual de usuario y obtener resultados o comportamientos que no coinciden con los anticipados por el documento. Por esta razón, la prueba de la documentación debería ser una parte importante de cualquier plan de pruebas del software.

La prueba de la documentación se puede enfocar en dos fases. La primera fase, *la revisión e inspección* (Capítulo 8), examina el documento para comprobar la claridad editorial. La segunda fase, *la prueba en vivo,* utiliza la documentación junto al uso del programa real.

Es sorprendente, pero la prueba en vivo de la documentación se puede enfocar usando técnicas análogas a muchos de los métodos de prueba de caja negra estudiados en la Sección 17.6. Se pueden emplear pruebas basadas en grafos para describir el empleo del programa; se pueden emplear el análisis de la partición equivalente o de los valores límites para definir varias clases de entradas e interacciones asociadas.

17.7.4. Prueba de sistemas de tiempo-real

La naturaleza asíncrona y dependiente del tiempo de muchas aplicaciones de tiempo real, añade un nuevo y potencialmente difícil elemento a la complejidad de las pruebas - e l tiempo—. El responsable del diseño de casos de prueba no sólo tiene que considerar los casos de prueba de caja blanca y de caja negra, sino también el tratamiento de sucesos (por ejemplo, procesamiento de interrupciones), la temporización de los datos y el paralelismo de las tareas (procesos) que manejan los datos. En muchas situaciones, los datos de prueba proporcionados al sistema de tiempo real cuando se encuentra en un determinado estado darán un proceso correcto, mientras que al proporcionárselos en otro estado llevarán a un error.


Sistemas de tiempo real.

Por ejemplo, un software de tiempo real que controla una moderna fotocopiadora puede aceptar interrupciones del operador (por ejemplo, el operador de la máquina pulsa teclas de control tales como «inicialización» u «oscurecimiento») sin error cuando la máquina se encuentra en el estado de hacer fotocopias (estado de «copia»). Esas mismas interrupciones del operador, cuando se producen estando la máquina en estado de «atasco», pueden producir un código de diagnóstico que indique la situación del atasco (un error).

Además, la estrecha relación que existe entre el software de tiempo real y su entorno de hardware también puede introducir problemas en la prueba. Las pruebas del software deben considerar el impacto de los fallos del hardware sobre el proceso del software. Puede ser muy difícil simular de forma realista esos fallos.

Todavía han de evolucionar mucho los métodos generales de diseño de casos de prueba para sistemas de tiempo real. Sin embargo, se puede proponer una estrategia en cuatro pasos:

> **Prueba de tareas.** El primer paso de la prueba de sistemas de tiempo real consiste en probar cada tarea independientemente. Es decir, se diseñan pruebas de caja blanca y de caja negra y se ejecutan para cada

tarea. Durante estas pruebas, cada tarea se ejecuta independientemente. La prueba de la tarea ha de descubrir errores en la lógica y en el funcionamiento, pero no los errores de temporización o de comportamiento.



El Forum de Discusión de la Prueba del Software presenta temas de interés a los profesionales que efectúan la prueba: **www.ondaweb.com /HyperNews/get.cgi/forums/sti.html**.

Prueba de comportamiento. Utilizando modelos del sistema creados con herramientas CASE, es posible simular el comportamiento del sistema en tiempo real y examinar su comportamiento como consecuencia de sucesos externos. Estas actividades de análisis pueden servir como base del diseño de casos de prueba que se llevan a cabo cuando se ha construido el software de tiempo real. Utilizando una técnica parecida a la partición equivalente (Sección 17.6.1), se pueden categorizar los sucesos (por ejemplo, interrupciones, señales de control, datos) para la prueba. Por ejemplo, los sucesos para la fotocopiadora pueden ser interrupciones del usuario (por ejemplo, reinicialización del contador), interrupciones mecánicas (por ejemplo, atasco del papel), interrupciones del sistema (por ejemplo, bajo nivel de tinta) y modos de fallo (por ejemplo, rodillo excesivamente caliente). Se prueba cada uno de esos sucesos individualmente y se examina el comportamiento del sistema ejecutable para detectar errores que se produzcan como consecuencia del proceso asociado a esos sucesos. Se puede comparar el comportamiento del modelo del sistema (desarrollado durante el análisis) con el software ejecutable para ver si existe concordancia. Una vez que se ha probado cada clase de sucesos, al sistema se le presentan sucesos en un orden aleatorio y con una frecuencia aleatoria. Se examina el comportamiento del software para detectar errores de comportamiento.

Prueba intertareas. Una vez que se han aislado los errores en las tareas individuales y en el comportamiento del sistema, la prueba se dirige hacia los errores relativos al tiempo. Se prueban las tareas asíncronas que se sabe que comunican con otras, con diferentes tasas de datos y cargas de proceso para determinar si se producen errores de sincronismo entre las tareas. Además, se prueban las tareas que se comunican mediante colas de mensajes o almacenes de datos, para detectar errores en el tamaño de esas áreas de almacenamiento de datos.

Prueba del sistema. El software y el hardware están integrados, por lo que se lleva a cabo una serie de pruebas completas del sistema (Capítulo 18) para intentar descubrir errores en la interfaz software/hardware.

La mayoría de los sistemas de tiempo real procesan interrupciones. Por tanto, es esencial la prueba del manejo de estos sucesos lógicos. Usando el diagrama estado-transición y la especificación de control (Capítulo 12), el responsable de la prueba desarrolla una lista de todas las posibles interrupciones y del proceso que ocurre como consecuencia de la interrupción. Se diseñan después pruebas para valorar las siguientes características del sistema:

- ¿Se han asignado y gestionado apropiadamente las prioridades de interrupción?
- ¿Se gestiona correctamente el procesamiento para todas las interrupciones?
- ¿Se ajusta a los requisitos el rendimiento (por ejemplo, tiempo de proceso) de todos los procedimientos de gestión de interrupciones?
- ¿Crea problemas de funcionamiento o rendimiento la llegada de un gran volumen de interrupciones en momentos críticos?

Además, se deberían probar las áreas de datos globales que se usan para transferir información como parte del proceso de una interrupción para valorar el potencial de generación de efectos colaterales.

RESUMEN

El principal objetivo del diseño de casos de prueba es obtener un conjunto de pruebas que tengan la mayor probabilidad de descubrir los defectos del software. Para llevar a cabo este objetivo, se usan dos categorías diferentes de técnicas de diseño de casos de prueba: prueba de caja blanca y prueba de caja negra.

Las pruebas de caja blanca se centran en la estructura de control del programa. Se obtienen casos de prueba que aseguren que durante la prueba se han ejecutado, por lo menos una vez, todas las sentencias del programa y que se ejercitan todas las condiciones lógicas. La prueba del camino básico, una técnica de caja blanca, hace uso de grafos de programa (o matrices de grafos) para obtener el conjunto de pruebas linealmente independientes que aseguren la total cobertura. La prueba de condiciones y del flujo de datos ejercita más aún la lógica del programa y la prueba de los bucles complementa a otras técnicas de caja blanca, proporcionando un procedimiento para ejercitar bucles de distintos grados de complejidad.

Heztel [HET84] describe la prueba de caja blanca como «prueba a pequeña escala». Esto se debe a que las pruebas de caja blanca que hemos considerado en este capítulo son típicamente aplicadas a pequeños componentes de programas (po ejemplo; módulos o pequeños grupos de módulos). Por otro lado, la prueba de caja negra amplía el enfoque y se puede denominar «prueba a gran escala».

Las pruebas de caja negra son diseñadas para validar los requisitos funcionales sin fijarse en el funcionamiento interno de un programa. Las técnicas de prueba de caja negra se centran en el ámbito de información de un programa, de forma que se proporcione una cobertura completa de prueba. Los métodos de prueba basados en grafos exploran las relaciones entre los objetos del programa y su comportamiento. La partición equivalente divide el campo de entrada en clases de datos que tienden a ejercitar determinadas funciones del software. El análisis de valores límite prueba la habilidad del programa para manejar datos que se encuentran en los límites aceptables. La prueba de la tabla ortogonal suministra un método sistemático y eficiente para probar sistemas con un número reducido de parámetros de entrada.

Los métodos de prueba especializados comprenden una amplia gama de capacidades del software y áreas de aplicación. Las interfaces gráficas de usuario, las arquitecturas cliente/servidor, la documentación y facilidades de ayuda, y los sistemas de tiempo real requieren directrices y técnicas especializadas **para** la prueba del software.

A menudo, los desarrolladores de software experimentados dicen que «la prueba nunca termina, simplemente se transfiere de usted (el ingeniero del software) al cliente: Cada vez que el cliente usa el programa, lleva a cabo una prueba.» Aplicando el diseño de casos de prueba, el ingeniero del software puede conseguir **una** prueba más completa y descubrir y corregir así el mayor número de errores antes de que comiencen las «pruebas del cliente».

REFERENCIAS

[BEI90] Beizer, B., *Software Testing Techniques,* 2.ª ed., Van Nostrand Reinhold, 1990.

[BEI95] Beizer, B., *Black-Box Testing,* Wiley, 1995.

[BRI87] Brilliant, S.S., J.C. Knight, y N.G. Levenson, «The Consistent Comparison Problem in N-Version Software», *ACM Software Engineering Notes,* vol. 12, n.° 1, enero 1987, pp. 29-34.

[DAV95] Davis, A., 201 *Principles of Software Development,* McGraw-Hill, 1995.

[DEU79] Deutsch, M., «Verification and Validation», *Software Engineering,* R. Jensen y C. Tonies (eds.), Prentice-Hall, 1979, pp. 329-408.

[FOS84] Foster, K.A., «Sensitive Test Data for Boolean Expressions», *ACM Software Engineering Notes,* vol. 9, n.° 2, Abril 1984, pp. 120-125.

[FRA88] Frankl, P.G., y E.J. Weyuker, «An Applicable Family of Data Flow Testing Criteria», *IEEE Trans. Software Engineering,* vol. 14, n.° 10, Octubre 1988, pp. 1483-1498.

[FRA93] Frankl, P.G., y S. Weiss, «An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow», *IEEE Trans. Software Engineering,* vol. 19, n.° 8, Agosto 1993, pp. 770-787.

[HET84] Hetzel, W., *The Complete Guide to Software Testing,* QED Information Sciences, Inc., Wellesley, MA, 1984.

[HOW82] Howden, W.E., «Weak Mutation Testing and the Completeness of Test Cases», *IEEE Trans. Software Engineering,* vol. SE-8, n.° 4, julio 1982, pp. 371-379.

[JON81] Jones, T.C., *Programming Productivity: Issues for the 80's,* IEEE Computer Society Press, 1981.

[KAN93] Kaner, C., J. Falk y H.Q. Nguyen, *Testing Computer Software,* 2.ª ed., Van Nostrand Reinhold, 1993.

[KNI89] Knight, J., y P. Ammann, «Testing Software Using Multiple Versions», Software Productivity Consortium, Report n.° 89029N, Reston, VA, Junio 1989.

[MCC76] McCabe, T., «A Software Complexity Measure», *IEEE Trans. Software Engineering,* vol. 2, Diciembre 1976, pp. 308-320.

[MYE79] Myers, G., *The Art of Software Testing,* Wiley, 1979.

[NTA88] Ntafos, S.C., «A comparison of Some Structural Testing Strategies», *IEEE Trans. Software Engineering,* vol. 14, n.° 6, Junio 1988, pp. 868-874.

[PHA89] Phadke, M.S., *Quality Engineering Using Robust Design,* Prentice Hall, 1989.

[PHA97] Phadke, M.S., «Planning Efficient Software Tests», *Crosstalk,* vol. 10, n.° 10, Octubre 1997, pp. 278-283.

[TAI87] Tai, K.C., y H.K. Su, «Test Generation for Boolean Expresions», *Proc. COMPSAC'87,* Octubre 1987, pp. 278-283.

[TAI89] Tai, K.C., «What to Do Beyond Branch Testing», *ACM Software Engineering Notes,* vol. 14, n.° 2, Abril 1989, pp. 58-61.

[WHI80] White, L.J., y E.I. Cohen, «A Domain Strategie for Program Testing», *IEEE Trans. Software Engineering,* vol. SE-6, n.° 5, Mayo 1980, pp. 247-257.

PROBLEMAS Y PUNTOS A CONSIDERAR

17.1. Myers [MYE79] usa el siguiente programa como autocomprobación de **su** capacidad para especificar una prueba adecuada: un programa lee tres valores enteros. Los tres valores se interpretan como representación de la longitud de los tres lados de un triángulo. El programa imprime un mensaje indicando si el triángulo es escaleno, isósceles o equilátero. Desarrolle un conjunto de casos de prueba que considere que probará de forma adecuada este programa.

17.2. Diseñe e implemente el programa especificado en el Problema 17.1 (con tratamiento de errores cuando sea necesario). Obtenga un grafo de flujo para el programa y aplique la prueba del camino básico para desarrollar casos de prueba que garanticen la comprobación de todas las sentencias del programa. Ejecute los casos y muestre sus resultados.

17.3. ¿Se le ocurren algunos objetivos de prueba adicionales que no se hayan mencionado en la Sección 17.1.1?

17.4. Aplique la técnica de prueba del camino básico a cualquiera de los programas que haya implementado en los Problemas 17.4 a 17.11.

17.5. Especifique, diseñe e implemente una herramienta de software que calcule la complejidad ciclomática para el lenguaje de programación que quiera. Utilice la matriz de grafos como estructura de datos operativa en el diseño.

17.6. Lea a Beizer [BEI90] y determine cómo puede ampliar el programa desarrollado en el Problema 17.5 para que incluya varios pesos de enlace. Amplíe la herramienta para que procese las probabilidades de ejecución o los tiempos de proceso de enlaces.

17.7. Use el enfoque de prueba de condiciones descrito en la Sección 17.5.1 para diseñar un conjunto de casos de prueba para el programa creado en el Problema 17.2.

17.8. Mediante el enfoque de prueba de flujo de datos descrito en la Sección 17.5.2, cree una lista de cadenas de definición-uso para el programa creado en el Problema 17.2.

17.9. Diseñe una herramienta automática que reconozca los bucles y los clasifique como indica la Sección 17.5.3.

17.10. Amplíe la herramienta descrita en el Problema 17.9 para que genere casos de prueba para cada categoría de bucle, cuando los encuentre. Será necesario llevar a cabo esta función de forma interactiva con el encargado de la prueba.

17.11. Dé por lo menos tres ejemplos en los que la prueba de caja negra pueda dar la impresión de que «todo está bien», mientras que la prueba de caja blanca pudiera descubrir errores. Indique por lo menos tres ejemplos en los que la prueba de caja blanca pueda dar la impresión de que «todo está bien», mientras que la prueba de caja negra pudiera descubrir errores.

17.12. ¿Podría una prueba exhaustiva (incluso si fuera posible para pequeños programas) garantizar que un programa es al 100 por 100 correcto?

17.13. Usando el método de la partición equivalente, obtenga un conjunto de casos de prueba para el sistema *HogarSeguro* descrito anteriormente en el libro.

17.14. Mediante el análisis de valores límite, obtenga un conjunto de casos de prueba para el sistema SSRB descrito en el Problema 12.13.

17.15. Investigue un poco y escriba un breve documento sobre el mecanismo de generación de tablas ortogonales para la prueba de datos.

17.16. Seleccione una IGU específica para un programa con el que esté familiarizado y diseñe una serie de pruebas para ejercitar la IGU.

17.17. Investigue en un sistema Cliente/Servidor que le sea familiar. Desarrolle un conjunto de escenarios de usuario y genere un perfil operacional para el sistema.

17.18. Pruebe un manual de usuario (o una facilidad de ayuda) de una aplicación que utilice frecuentemente. Encuentre al menos un error en la documentación.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

La ingeniería del software presenta tanto desafíos técnicos como de gestión. Los libros de Black (*Managing the Testing Process*, Microsoft Press, 1999), Dustin, Rashka y Paul (*Test process Improvement: Step-By-Step Guide to Structured Testing*, Addison-Wesley, 1999), Peny (*Surviving the Top Ten Challenges of Software Testing: A People-Oriented Approach*, Dorset House, 1997), y Kit y Finzi (*Software Testing in the Real World: Improving the Process*, Addison-Wesley, 1995) orientan sobre las necesidades de gestión y de procesos.

Para aquellos lectores que deseen' información adicional sobre la tecnología de prueba del software, existen varios libros excelentes. Kaner, Nguyen y Falk (*Testing Computer Software*, Wiley, 1999), Hutcheson (*Sofiare Testing Methods and Metrics: The Most Important Test*, McGraw-Hill, 1997), Marick (*The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing*, Prentice-Hall, 1995), Jorgensen (*Software Testing: A Craftman's Approach*, CRC Press, 1995) presentan estudios sobre los métodos y estrategias de prueba.

Myers [MYE79] permanece como un texto clásico, cubriendo con considerable detalle las técnicas de caja negra. Beizer [BEI90] da una amplia cobertura de las técnicas de caja blanca, introduciendo cierto nivel de rigor matemático que a menudo se echa en falta en otros tratados sobre la prueba. Su último libro [BEI95] presenta un tratado conciso de métodos importantes. Perry (*Effective Methods for Software Testing*, Wiley-QED, 1995), y Freeman y Voas (*Sofiare Assesment: Reliability, Sfety, Testability*, Wiley, 1995) presentan buenas introducciones a las estrategias y tácticas de las pruebas. Mosley (*The Handbook of MIS Application Software Testing*, Prentice-Hall, 1993) estudia aspectos de las pruebas para grandes sistemas de información, y Marks (*Testing Very Big Systems*, McGraw-Hill, 1992) estudia los aspectos especiales que deben considerarse cuando se prueban grandes sistemas de programación.

La prueba del software es un recurso en continua actividad. Es por esta razón por lo que muchas organizaciones automatizan parte de los procesos de prueba. Los libros de Dustin, Rashka y Poston (*Automated Software Testing: Introduction, Management, and Performance*, Addison-Wesley, 1999) y Poston (*Automating Specification-Based Software Testing*, IEEE Computer Society, 1996) hablan sobre herramientas, estrategias y métodos para automatizar la prueba. Una excelente fuente de información sobre herramientas automatizadas de prueba la encontramos en *Testing Tools Reference Guide* (Software Quality Engineering, Inc., Jacksonville, FL, actualizada anualmente). Este directorio contiene descripciones de cientos de herramientas de prueba, clasificadas por tipo de prueba, plataforma hardware y soporte software.

Algunos libros tratan los métodos y estrategias de prueba en áreas de aplicación especializada. Gardiner (*Testing Safety-Related Software: A Practica1 Handbook*, Springer Verlag, 1999) ha editado un libro que trata la prueba en sistemas de seguridad crítica. Mosley (*Client/Server Software Testing on the Desk Top and the Web*, Prentice Hall, 1999) trata la prueba para clientes, servidores y componentes en red. Rubin (*Handbook of Usability Testing*, Wiley, 1994) ha escrito una guía útil para lo que necesitan manejar interfaces humanas.

Una amplia variedad de fuentes de información sobre pruebas del software y elementos relacionados están disponibles en internet. Una lista actualizada de referencias a páginas web que son relevantes sobre los conceptos de prueba, métodos y estrategias se pueden encontrar en **http://www.pressman5.com**.

.

CAPÍTULO

18 ESTRATEGIAS DE PRUEBA DE SOFTWARE

UNA estrategia de prueba del software integra las técnicas de diseño de casos de prueba en una serie de pasos bien planificados que dan como resultado una correcta construcción del software. La estrategia proporciona un mapa que describe los pasos que hay que llevar a cabo como parte de la prueba, cuándo se deben planificar y realizar esos pasos, y cuánto esfuerzo, tiempo y recursos se van a requerir. Por tanto, cualquier estrategia de prueba debe incorporar la planificación de la prueba, el diseño de casos de prueba, la ejecución de las pruebas y la agrupación y evaluación de los datos resultantes.

Una estrategia de prueba del software debe ser suficientemente flexible para promover la creatividad y la adaptabilidad necesarias para adecuar la prueba a todos los grandes sistemas basados en software. Al mismo tiempo, la estrategia debe ser suficientemente rígida para promover un seguimiento razonable de la planificación y la gestión a medida que progresa el proyecto. Shooman [SHO83] trata estas cuestiones:

> En muchos aspectos, la prueba es un proceso individualista, y el número de tipos diferentes de pruebas varía tanto como los diferentes enfoques de desarrollo. Durante muchos años, nuestra Única defensa contra los errores de programación era un cuidadoso diseño y la propia inteligencia del programador. Ahora nos encontramos en una era en la que las técnicas modernas de diseño [y las revisiones técnicas formales] nos ayudan a reducir el número de errores iniciales que se encuentran en el código de forma inherente. De forma similar, los diferentes métodos de prueba están empezando a agruparse en varias filosofías y enfoques diferentes.

VISTAZO RÁPIDO

¿Qué es? El diseño efectivo de casos de prueba (Capítulo 17) es importante, pero también lo es la estrategia para su utilización. ¿Se deberá desarrollar un plan formal para estas pruebas?. ¿Se deberá probar el programa íntegramente o ejecutar pruebas solamente sobre una parte pequeña del mismo? ¿Se deberán ejecutar pruebas de regresión cuando se añadan nuevos componentes al sistema? ¿Cuándo se deberá involucrar al cliente? Estas y otras muchas cuestiones serán contestadas cuando desarrolles una estrategia de prueba del software.

¿Quién lo hace? El jefe del proyecto, los ingenieros del software y los especialistas en pruebas.

¿Por qué es importante? En el proyecto, la prueba a veces requiere más esfuerzo que cualquier otra actividad de la ingeniería del software. Si se efectúa sin un plan, el tiempo se desaprovecha y el esfuerzo es consumido innecesariamente y, en el peor de los casos, los errores inadvertidos quedarán sin detectar. Por tanto, parece razonable establecer una estrategia sistemática para probar el software.

¿Cuáles son los pasos? La prueba comienza por «lo pequeño» y progresa hacia «lo grande». Por esta razón, debemos comenzar las primeras pruebas sobre el componente elemental y aplicar sobre él pruebas de caja blanca y de caja negra para descubrir errores en la lógica y en la funcionalidad del programa. Después de que los componentes elementales hayan sido aprobados, procederemos a su integración. Las pruebas se efectuarán conforme el software se vaya construyendo.

Finalmente, una serie de pruebas de alto nivel serán ejecutadas una vez el programa esté totalmente preparado para su operatividad. Estas pruebas están diseñadas para descubrir errores en los requisitos.

¿Cuál es el producto obtenido? El equipo de trabajo que desarrolla el software documenta la *especificación de la prueba* basándose en la definición del plan que establece la estrategia general y del procedimiento que específicamente describe los pasos a seguir y las pruebas a realizar.

¿Cómo puedo estar seguro de que lo he hecho correctamente? La revisión de la *especificación de la prueba* es previa a la realización de la prueba. Se debe valorar la completitud de los casos de prueba y de las tareas de la prueba. Un plan y un procedimiento de prueba efectivo permitirá una construcción ordenada del software y el descubrimiento de errores en cada etapa del proceso de construcción.

Estas «filosofías y enfoques» constituyen lo que nosotros llamaremos estrategia. En el Capítulo 17 se presentó la tecnología de prueba del software[1]. En este capítulo centraremos nuestra atención en las estrategias de prueba del software.

[1] Las pruebas de sistemas orientados a objetos se estudian en el Capítulo 23.

18.1 UN ENFOQUE ESTRATÉGICO PARA LAS PRUEBAS DEL SOFTWARE

Las pruebas son un conjunto de actividades que se pueden planificar por adelantado y llevar a cabo sistemáticamente. Por esta razón, se debe definir en el proceso de la ingeniería del software una *plantilla* para las pruebas del software: un conjunto de pasos en los que podamos situar los métodos específicos de diseño de casos de prueba.

Se han propuesto varias estrategias de prueba del software en distintos libros. Todas proporcionan al ingeniero del software una plantilla para la prueba y todas tienen las siguientes características generales:

- Las pruebas comienzan a nivel de módulo[2] y trabajan «hacia fuera», hacia la integración de todo el sistema basado en computadora.
- Según el momento, son apropiadas diferentes técnicas de prueba.
- La prueba la lleva a cabo el responsable del desarrollo del software y (para grandes proyectos) un grupo independiente de pruebas.
- La prueba y la depuración son actividades diferentes, pero la depuración se debe incluir en cualquier estrategia de prueba.

Una estrategia de prueba del software debe incluir pruebas de bajo nivel que verifiquen que todos los pequeños segmentos de código fuente se han implementado correctamente, así como pruebas de alto nivel que validen las principales funciones del sistema frente a los requisitos del cliente. Una estrategia debe proporcionar una guía al profesional y proporcionar un conjunto de hitos para el jefe de proyecto. Debido a que los pasos de la estrategia de prueba se dan a la vez cuando aumenta la presión de los plazos fijados, se debe poder medir el progreso y los problemas deben aparecer lo antes posible.



Información útil sobre las estrategias de prueba del software es suministrada en el informe sobre la Prueba del Software en: **www.ondaweb.com/sti/newsltr.htm**

18.1.1. Verificación y validación

La prueba del software es un elemento de un tema más amplio que, a menudo, es conocido como verificación y validación (**V&V**). La *verificación* se refiere al conjunto de actividades que aseguran que el software implementa correctamente una función específica. La *validación* se refiere a un conjunto diferente de actividades que aseguran que el software construido se ajusta a los requisitos del cliente. Bohem [BOE81] lo define de otra forma:

> *Verificación:* «¿Estamos construyendo el producto correctamente?»
>
> *Validación:* «¿Estamos construyendo el producto correcto?»

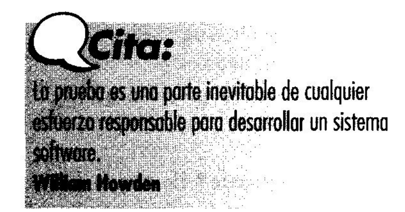
La definición de V&V comprende muchas de las actividades a las que nos hemos referido como garantía de calidad del software (SQA*).

La verificación y la validación abarcan una amplia lista de actividades SQA que incluye: revisiones técnicas formales, auditorías de calidad y de configuración, monitorización de rendimientos, simulación, estudios de factibilidad, revisión de la documentación, revisión de la base de datos, análisis algorítmico, pruebas de desarrollo, pruebas de validación y pruebas de instalación [WAL89]. A pesar de que las actividades de prueba tienen un papel muy importante en V&V, muchas otras actividades son también necesarias.

Referencia cruzada

Las actividades SQA son estudiadas en detalle en el Capítulo 8.

Las pruebas constituyen el último bastión desde el que se puede evaluar la calidad y, de forma más pragmática, descubrir los errores. Pero las pruebas no deben ser vistas como una red de seguridad. Como se suele decir: «No se puede probar la calidad. Si no está ahí antes de comenzar la prueba, no estará cuando se termine.» La calidad se incorpora en el software durante el proceso de ingeniería del software. La aplicación adecuada de los métodos y de las herramientas, las revisiones técnicas formales efectivas y una sólida gestión y medición, conducen a la calidad, que se confirma durante las pruebas.



Miller [MIL77] relaciona la prueba del software con la garantía de calidad al establecer que «la motivación subyacente de la prueba de programas es confirmar la calidad del software con métodos que se pueden aplicar de forma económica y efectiva, tanto a grandes como a pequeños sistemas».

2 Para los sistemas orientados a objetos, las pruebas empiezan a nivel de clase o de objeto. Vea más detalles en el Capítulo 23.

* Por lo habitual de su utilización mantenemos el termino ingles SQA (Software Quality Assurance).

18.1.2. Organización para las pruebas del software

En cualquier proyecto de software existe un conflicto de intereses inherente que aparece cuando comienzan las pruebas. Se pide a la gente que ha construido el software que lo pruebe. Esto parece totalmente inofensivo: después de todo, ¿quién puede conocer mejor un programa que los que lo han desarrollado? Desgraciadamente, esos mismos programadores tienen un gran interés en demostrar que el programa está libre de errores, que funciona de acuerdo con las especificaciones del cliente y que estará listo de acuerdo con los plazos y el presupuesto. Cada uno de estos intereses se convierte en inconveniente a la hora de encontrar errores a lo largo del proceso de prueba.

Desde un punto de vista psicológico, el análisis y el diseño del software (junto con la codificación) son tareas constructivas. El ingeniero del software crea un programa de computadora, su documentación y sus estructuras de datos asociadas. Al igual que cualquier constructor, el ingeniero del software está orgulloso del edificio que acaba de construir y se enfrenta a cualquiera que intente sacarle defectos.

Cuando comienza la prueba, aparece una sutil, aunque firme intención de «romper» lo que el ingeniero del software ha construido. Desde el punto de vista del constructor, la prueba se puede considerar (psicológicamente) *destructiva*. Por tanto, el constructor anda con cuidado, diseñando y ejecutando pruebas que demuestren que el programa funciona, en lugar de detectar errores. Desgraciadamente, los errores seguirán estando. Y si el ingeniero del software no los encuentra, ¡el cliente **si** lo hará!

A menudo, existen ciertos malentendidos que se pueden deducir equivocadamente de la anterior discusión: (1) el responsable del desarrollo no debería entrar en el proceso de prueba; (2) el software debe ser «puesto a salvo» de extraños que puedan probarlo de forma despiadada; **(3)** los encargados de la prueba sólo aparecen en el proyecto cuando comienzan las etapas de prueba. Todas estas frases son incorrectas.

El responsable del desarrollo del software siempre es responsable de probar las unidades individuales (módulos) del programa, asegurándose de que cada una lleva a cabo la función para la que fue diseñada. En muchos casos, también se encargará de la prueba de integración, el paso de las pruebas que lleva a la construcción (y prueba) de la estructura total del sistema. Sólo una vez que la arquitectura del software esté completa entra en juego un grupo independiente de prueba.



Un grupo independiente de prueba no tiene el ((conflicto de intereses) que tienen los desarrolladores del software.

El papel del grupo independiente de prueba (GIP) es eliminar los inherentes problemas asociados con el hecho de permitir al constructor que pruebe lo que ha construido. Una prueba independiente elimina el conflicto de intereses que, de otro modo, estaría presente. Después de todo, al personal del equipo que forma el grupo independiente se le paga para que encuentre errores.

Sin embargo, el responsable del desarrollado del software no entrega simplemente el programa al GIP y se desentiende. El responsable del desarrollado y el GIP trabajan estrechamente a lo largo del proyecto de software para asegurar que se realizan pruebas exhaustivas. Mientras se realiza la prueba, el desarrollador debe estar disponible para corregir los errores que se van descubriendo.

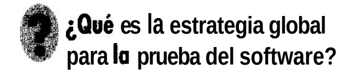


Si no existe un GIP en tu organización, tendrás que asumir su punto de vista. Cuando pruebes, intenta destrozar el software.

El GIP es parte del equipo del proyecto de desarrollo de software en el sentido de que se ve implicado durante el proceso de especificación y sigue implicado (planificación y especificación de los procedimientos de prueba) a lo largo de un gran proyecto. Sin embargo, en muchos casos, el GIP informa a la organización de garantía de calidad del software, consiguiendo de este modo un grado de independencia que no sería posible si fuera una parte de la organización de desarrollo del software.

18.1.3. Una estrategia de prueba del software

El proceso de ingeniería del software se puede ver como una espiral, como se ilustra en la Figura 18.1. Inicialmente, la ingeniería de sistemas define el papel del software y conduce al análisis de los requisitos del software, donde se establece el dominio de información, la función, el comportamiento, el rendimiento, las restricciones y los criterios de validación del software. Al movemos hacia el interior de la espiral, llegamos al diseño y, por último, a la codificación. Para desarrollar software de computadora, damos vueltas en espiral a través de una serie de flujos o líneas que disminuyen el nivel de abstracción en cada vuelta,



También se puede ver la estrategia para la prueba del software en el contexto de la espiral (Fig. 18.1). La *prueba* de unidad comienza en el vértice de la espiral y se centra en cada unidad del software, tal como está implementada en código fuente. La prueba avanza, al movernos hacia fuera de la espiral, hasta llegar a la prueba ***de*** integración, donde el foco de atención es el diseño y la construcción de la arquitectura del software. Dando otra

vuelta por la espiral hacia fuera, encontramos la *prueba de validación,* donde se validan los requisitos establecidos como parte del análisis de requisitos del software, comparándolos con el sistema que ha sido construido. Finalmente, llegamos a la *prueba del sistema*, en la que se prueban como un todo el software y otros elementos del sistema. Para probar software de computadora nos movemos hacia fuera por una espiral que, a cada vuelta, aumenta el alcance de la prueba.

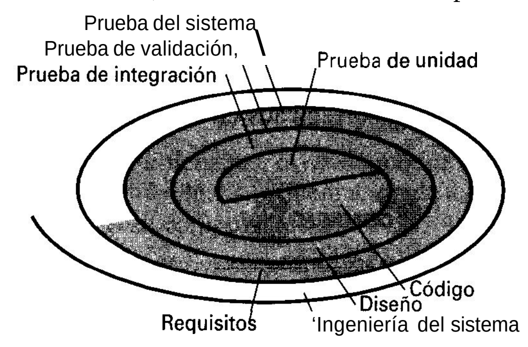


FIGURA 18.1. Estrategia de prueba.

Si consideramos el proceso desde el punto de vista procedimental, la prueba, en el contexto de la ingeniería del software, realmente es una serie de cuatro pasos que se llevan a cabo secuencialmente. Esos pasos se muestran en la Figura 18.2. Inicialmente, la prueba se centra en cada módulo individualmente, asegurando que funcionan adecuadamente como una unidad. De ahí el nombre de *prueba de unidad.* La prueba de unidad hace un uso intensivo de las técnicas de prueba de caja blanca, ejercitando caminos específicos de la estructura de control del módulo para asegurar **un** alcance completo **y** una detección máxima de errores. A continuación, se deben ensamblar o integrar los módulos para formar el paquete de software completo. La *prueba de integración* se dirige a todos los aspectos asociados con el doble problema de verificación **y** de construcción del programa. Durante la integración, las técnicas que más prevalecen son las de diseño de casos de prueba de caja negra, aunque se pueden llevar a cabo algunas pruebas de caja blanca con el fin de asegurar que se cubren los principales caminos de control. Después de que el software se ha integrado (construido), se dirigen un conjunto de *pruebas de alto nivel.* Se deben comprobar los criterios de validación (establecidos durante el análisis de requisitos). La *prueba de validación* proporciona una seguridad final de que el software satisface todos los requisitos funcionales, de comportamiento y de rendimiento. Durante la validación se usan exclusivamente técnicas de prueba de caja negra.

Referencia cruzada

Los técnicas de pruebo de cojo blanca y cojo negra se estudian en el Capítulo 17.

El último paso de prueba de alto nivel queda fuera de los límites de la ingeniería del software, entrando en el más amplio contexto de la ingeniería de sistemas de computadora.

El software, una vez validado, se debe combinar con otros elementos del sistema (por ejemplo, hardware, gente, bases de datos). La *prueba del sistema* verifica que cada elemento encaja de forma adecuada y que se alcanza la funcionalidad y el rendimiento del sistema total.

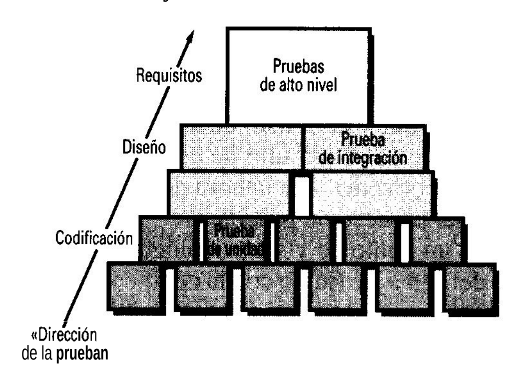


FIGURA 18.2. Etapas en la prueba del software.

18.1.4. Criterios para completar la prueba

Cada vez que se tratan las pruebas del software surge una pregunta clásica: ¿Cuando hemos terminado las pruebas?, ¿cómo sabemos que hemos probado lo suficiente? Desgraciadamente, no hay una respuesta definitiva a esta pregunta, pero hay algunas respuestas prácticas **y** nuevos intentos de base empírica.



Una respuesta a la pregunta anterior es: «La prueba nunca termina, ya que el responsable del desarrollo del software carga o pasa el problema al cliente.» Cada vez que el cliente/usuario ejecuta un programa de computadora, dicho programa se está probando con un nuevo conjunto de datos. Este importante hecho subraya la importancia de otras actividades de garantía de calidad del software. Otra respuesta (algo cínica, pero sin embargo cierta) es: «Se termina la prueba cuando se agota el tiempo o el dinero disponible para tal efecto.»

Aunque algunos profesionales se sirvan de estas respuestas como argumento, un ingeniero del software necesita un criterio más riguroso para determinar cuando se ha realizado la prueba suficiente. Musa y Ackerman [MUS89] sugieren una respuesta basada en un criterio estadístico: «No, no podemos tener la absoluta certeza de que el software nunca fallará, pero en base a un modelo estadístico de corte teórico y validado experimentalmente, hemos realizado las pruebas suficientes para decir, con un 95 por 100 de certeza, que la probabilidad de funcionamiento libre de fallo de 1.000 horas de CPU, en un entorno definido de forma probabilística, es al menos 0,995.»

Mediante el modelado estadístico **y** la teoría de fiabilidad del software, se pueden desarrollar modelos de fallos

del software (descubiertos durante las pruebas) como una función del tiempo de ejecución. Una versión del modelo de fallos, denominado *modelo logarítmico de Poisson de tiempo de ejecución,* toma la siguiente forma:

$$f(t) = (1/p)\ ln\ [(l_0 pt + 1)] \qquad (18.1)$$

donde

$f(t)$ número acumulado de fallos que se espera que se produzcan una vez que se ha probado el software durante una cierta cantidad de tiempo de ejecución t,

l_0 la intensidad de fallos inicial del software (fallos por unidad de tiempo) al principio de la prueba

p la reducción exponencial de intensidad de fallo a medida que se encuentran los errores y se van haciendo las correcciones.

La intensidad de fallos instantánea, $l(t)$ se puede obtener mediante la derivada de $f(t)$:

$$l(t) = l_0 / (l_0 pt + 1) \qquad (18.2)$$

Mediante la relación de la ecuación (18.2), los que realizan las pruebas pueden predecir la disminución de errores a medida que estas avanzan. La intensidad de error real se puede trazar junto a la curva predecida (Fig. **18.3).** Si los datos reales recopilados durante la prueba y el modelo logarítmico de Poisson de tiempo de ejecución están razonablemente cerca unos de otros, sobre un número de puntos de datos, el modelo se puede usar para predecir el tiempo de prueba total requerido para alcanzar una intensidad de fallos aceptablemente baja.

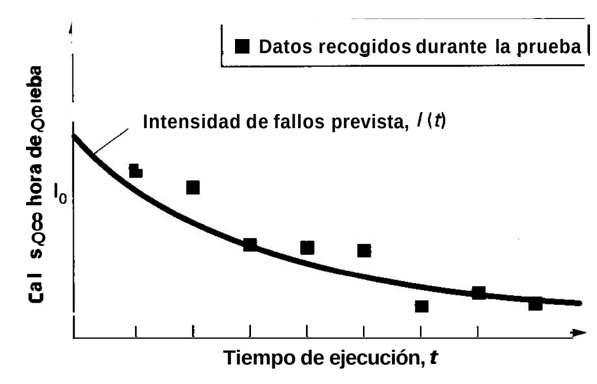
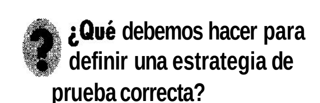


FIGURA 18.3. Intensidad de fallos en función del tiempo de ejecución.

Mediante la agrupación de métricas durante la prueba del software y haciendo uso de los modelos de fiabilidad del software existentes, es posible desarrollar directrices importantes para responder a la pregunta: ¿cuándo terminamos la prueba? No hay duda que todavía queda mucho trabajo por hacer antes de que se puedan establecer reglas cuantitativas para la prueba, pero los enfoques empíricos que existen actualmente son considerablemente mejores que la pura intuición.

18.2 ASPECTOS ESTRATÉGICOS

Más adelante, en este capítulo, exploramos una estrategia sistemática para la prueba del software. Pero incluso la mejor estrategia fracasará si no se tratan una serie **de** aspectos invalidantes. Tom Gilb [GIL95] plantea que se deben abordar los siguientes puntos si se desea implementar con éxito una estrategia de prueba del software:


¿Qué debemos hacer para definir una estrategia de prueba correcta?

Especificar los requisitos del producto de manera cuantificable mucho antes de que comiencen las pruebas. Aunque el objetivo principal de la prueba es encontrar errores, una buena estrategia de prueba también evalúa otras características de la calidad, tales como la portabilidad, facilidad de mantenimiento y facilidad de uso (Capítulo 19). Todo esto debería especificarse de manera que sea medible para que los resultados de la prueba no sean ambiguos.

Establecer los objetivos de la prueba de manera explícita. Se deberían establecer en términos medibles los objetivos específicos de la prueba. Por ejemplo, la efectividad de la prueba, la cobertura de la prueba, tiempo medio de fallo, el coste para encontrar y arreglar errores, densidad de fallos remanente o frecuencia de ocurrencia, y horas de trabajo por prueba de regresión deberían establecerse dentro de la planificación de la prueba [GIL95].

Comprender qué usuarios van a manejar el software y desarrollar un perfil para cada categoría de usuario. Usar casos que describan el escenario de interacción para cada clase de usuario pudiendo reducir el esfuerzo general de prueba concentrando la prueba en el empleo real del producto.

Referencia cruzada

Los casos de uso describen un escenario para usar el software **y se** estudian en el Capítulo 11.

Desarrollar un plan de prueba que haga hincapié en la «prueba de ciclo rápido». Gilb [GIL95] recomienda que un equipo de ingeniería del software «aprenda a probar en ciclos rápidos (2 por 100 del esfuerzo del proyecto) de incrementos de funcionalidad y/o mejora de la calidad Útiles para el cliente, y que se puedan probar sobre el terreno». La realimentación generada por estas pruebas de ciclo rápido puede usarse para controlar los niveles de calidad y las correspondientes estrategias de prueba.



La prueba sólo de los requisitos percibidos por el usuario final es semejante a la inspección de una construcción basada en el trabajo realizado por un decorador sobre el gasto en cimientos, vigas y fontanería.
Boris Beizer

Construir un software «robusto» diseñado para probarse a sí mismo. El software debería diseñarse de manera que use técnicas de depuración antierrores (Sección 18.3.1). **Es** decir, el software debería ser capaz de diagnosticar ciertas clases de errores. Además, el diseño debería incluir pruebas automatizadas y pruebas de regresión.

Usar revisiones técnicas formales efectivas comofiltro antes de la prueba. Las revisiones técnicas formales (Capítulo **8)** pueden ser tan efectivas como las pruebas en el descubrimiento de errores. **Por** este motivo, las revisiones pueden reducir la cantidad de esfuerzo de prueba necesaria para producir software de alta calidad.

Llevar a cabo revisiones técnicasformales para evaluar la estrategia de prueba y los propios casos de prueba. Las revisiones técnicas formales pueden descubrir inconsistencias, omisiones y errores claros en el enfoque de la prueba. Esto ahorra tiempo y también mejora la calidad del producto.

Desarrollar un enfoque de mejora continua al proceso de prueba. Debería medirse la estrategia de prueba. Las métricas agrupadas durante la prueba deberían usarse como parte de un enfoque estadístico de control del proceso para la prueba del software.

18.3 PRUEBA DE UNIDAD

La prueba de unidad centra el proceso de verificación en la menor unidad del diseño del software: el componente software o módulo.

18.3.1. Consideraciones sobre la prueba de unidad

Las pruebas que se dan como parte de la prueba de unidad están esquemáticamente ilustradas en la Figura 18.4. Se prueba la interfaz del módulo para asegurar que la información fluye de forma adecuada hacia y desde la unidad de programa que está siendo probada. Se examinan las estructuras de datos locales para asegurar que los datos que se mantienen temporalmente conservan su integridad durante todos los pasos de ejecución del algoritmo. Se prueban las condiciones límite para asegurar que el módulo funciona correctamente en los límites establecidos como restricciones de procesamiento. Se ejercitan todos los caminos independientes (caminos básicos) de la estructura de control con el fin de asegurar que todas las sentencias del módulo se ejecutan por lo menos una vez. **Y,** finalmente, se prueban todos los caminos de manejo de errores.

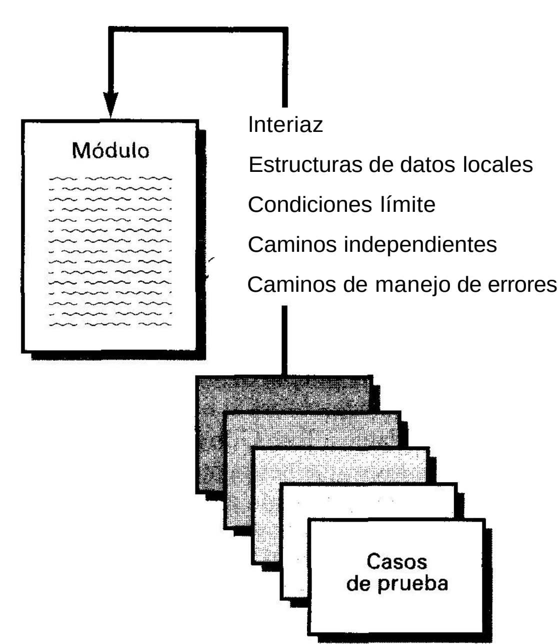
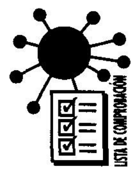


FIGURA 18.4. Prueba de unidad.



Prueba de Unidad.

Antes de iniciar cualquier otra prueba es preciso probar el flujo de datos de la interfaz del módulo. Si los datos no entran correctamente, todas las demás pruebas no tienen sentido. Además de las estructuras de datos locales, durante la prueba de unidad se debe comprobar (en la medida de lo posible) el impacto de los datos globales sobre el módulo.

Durante la prueba de unidad, la comprobación selectiva de los caminos de ejecución es una tarea esencial. Se deben diseñar casos de prueba para detectar errores debidos a cálculos incorrectos, comparaciones incorrectas o flujos de control inapropiados. Las pruebas del camino básico y de bucles son técnicas muy efectivas para descubrir una gran cantidad de errores en los caminos.



Entre los errores más comunes en los cálculos están: (1) precedencia aritmética incorrecta o mal interpretada;

(2) operaciones de modo mezcladas; (3) inicializaciones incorrectas; (4) falta de precisión; (5) incorrecta representación simbólica de una expresión. Las comparaciones y el flujo de control están fuertemente emparejadas (por ejemplo, el flujo de control cambia frecuentemente tras una comparación). Los casos de prueba deben descubrir errores como: (1) comparaciones ente tipos de datos distintos; (2) operadores lógicos o de precedencia incorrectos; (3) igualdad esperada cuando los errores de precisión la hacen poco probable; (4) variables o comparaciones incorrectas; (5) terminación de bucles inapropiada o inexistente; (6) fallo de salida cuando se encuentra una iteración divergente, y (7) variables de bucles modificadas de forma inapropiada.

Un buen diseño exige que las condiciones de error sean previstas de antemano y que se dispongan unos caminos de manejo de errores que redirijan o terminen de una forma limpia el proceso cuando se dé un error. Yourdon [YOU75] llama a este enfoque *antipurgado*. Desgraciadamente, existe una tendencia a incorporar la manipulación de errores en el software y así no probarlo nunca. Como ejemplo, sirve una historia real:

> **Mediante un contrato se desarrolló un importante sistema de diseño interactivo. En un módulo de proceso de transacciones, un bromista puso el siguiente mensaje de manipulación de error que aparecía tras una serie de pruebas condicionales que invocaban varias ramificaciones del flujo de control: ¡ERROR! NO HAY FORMA DE QUE VD. LLEGUE HASTA AQUÍ. ¡Este «mensaje de error» fue descubierto por un cliente durante la fase de puesta a punto!**



Asegura que tu diseño de pruebas ejecuta todos los caminos pura encontrar errores. Si no lo haces, el camino puede fallar al ser invocado, provocando una situación incierta.

Entre los errores potenciales que se deben comprobar cuando se evalúa la manipulación de errores están:

1. Descripción ininteligible del error.
2. El error señalado no se corresponde con el error encontrado.
3. La condición de error hace que intervenga el sistema antes que el mecanismo de manejo de errores.
4. El procesamiento de la condición excepcional es incorrecto.
5. La descripción del error no proporciona suficiente información para ayudar en la localización de la causa del error.

La prueba de límites es la última (y probablemente, la más importante) tarea del paso de la prueba de unidad. El software falla frecuentemente en sus condiciones límite. Es decir, con frecuencia aparece un error cuando se procesa el elemento n-ésimo de un array n-dimensional, cuando se hace la i-ésima repetición de un bucle de i pasos o cuando se encuentran los valores máximo o mínimo permitidos. Los casos de prueba que ejerciten las estructuras de datos, el flujo de control y los valores de los datos por debajo y por encima de los máximos y los mínimos son muy apropiados para descubrir estos errores.

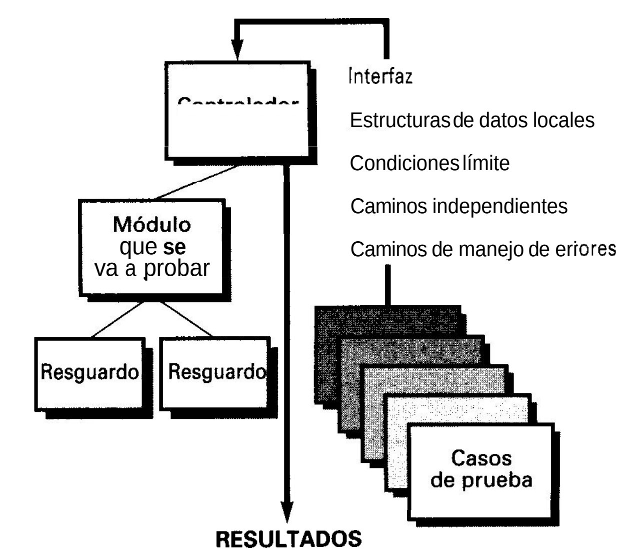


FIGURA 18.5. Entorno para la prueba de unidad.

18.3.2. Procedimientos de Prueba de Unidad

Debido a que un componente no es un programa independiente, se debe desarrollar para cada prueba de unidad un software que controle y/o resguarde. En la Figura 18.5 se ilustra el entorno para la prueba de unidad. En la mayoría de las aplicaciones, un *controlador* no es más que un «programa principal» que acepta los datos del caso de prueba, pasa estos datos al módulo (a ser probado) e imprime los resultados importantes. Los resguardos sirven para reemplazar módulos que están subordinados (llamados por) el componente que hay que probar. Un resguardo o un «subprograma simulado» usa la interfaz del módulo subordinado, lleva a cabo una mínima manipulación de datos, imprime una verificación de entrada y devuelve control al módulo de prueba que lo invocó.



Hay ocasiones en que no dispones de los recursos para hacer una prueba unitaria completa. En esta situación, selecciona los módulos críticos y aquellos con alta complejidad ciclomática y realiza sobre ellos la prueba unitaria.

Los controladores y los resguardos son una sobrecarga de trabajo. Es decir, ambos son software que debe desarrollarse (normalmente no se aplica un diseño formal) pero que no se entrega con el producto de software final. Si los controladores y resguardos son sencillos, el trabajo adicional es relativamente pequeño. Desgraciadamente, muchos componentes no pue-

den tener una adecuada prueba unitaria con un «sencillo» software adicional. En tales casos, la prueba completa se pospone hasta que se llegue al paso de prueba de integración (donde también se usan controladores o resguardos).

La prueba de unidad se simplifica cuando se diseña un módulo con un alto grado de cohesión. Cuando un módulo sólo realiza una función, se reduce el número de casos de prueba y los errores se pueden predecir y descubrir más fácilmente.

18.4 PRUEBA DE INTEGRACIÓN[3]

Un neófito del mundo del software podría, una vez que se les ha hecho la prueba de unidad a todos los módulos, cuestionar de forma aparentemente legítima lo siguiente: «Si todos funcionan bien por separado, ¿por **qué** dudar de que funcionen todos juntos?» Por supuesto, el problema es «ponerlos juntos» (interacción). Los datos se pueden perder en una interfaz; un módulo puede tener un efecto adverso e inadvertido sobre otro; las subfunciones, cuando se combinan, pueden no producir la función principal deseada; la imprecisión aceptada individualmente puede crecer hasta niveles inaceptables; y las estructuras de datos globales pueden presentar problemas. Desgraciadamente, la lista sigue y sigue.

La prueba de integración es una técnica sistemática para construir la estructura del programa mientras que, al mismo tiempo, se llevan a cabo pruebas para detectar errores asociados con la interacción. El objetivo es coger los módulos probados mediante la prueba de unidad y construir una estructura de programa que esté de acuerdo con lo que dicta el diseño.



Efectuar una integración ***big*** *bag es una estrategia vaga que* ***está*** *condenada al fracaso. la prueba de integración* ***deberá*** *ser conducida incrementalmente.*

A menudo hay una tendencia a intentar una integración no incremental; es decir, a construir el programa mediante un enfoque de *«big bang»*. Se combinan todos los módulos por anticipado. Se prueba todo el programa en conjunto. ¡Normalmente se llega al caos! Se encuentran un gran conjunto de errores. La corrección se hace difícil, puesto que es complicado aislar las causas al tener delante el programa entero en toda su extensión. Una vez que se corrigen esos errores aparecen otros nuevos y el proceso continúa en lo que parece ser un ciclo sin fin.

La integración incremental es la antítesis del enfoque del «big bang». El programa se construye y se prueba en pequeños segmentos en los que los errores son más fáciles de aislar y de corregir, es más probable que se puedan probar completamente las interfaces y se puede aplicar un enfoque de prueba sistemática. En las siguientes secciones se tratan varias estrategias de integración incremental diferentes.

18.4.1. Integración descendente

La prueba de integración descendente es un planteamiento incremental a la construcción de la estructura de programas. Se integran los módulos moviéndose hacia abajo por la jerarquía de control, comenzando por el módulo de control principal (programa principal). Los módulos subordinados (subordinados de cualquier modo) al módulo de control principal se van incorporando en la estructura, bien de forma *primero-en-profundidad*, o bien de forma *primero-en-anchura.*



Cuando desarrollas una planificación detallada del ***proyecto*** *debes considerar la manera en que la integración se va a realizar, de* ***forma*** *que los componentes estén disponibles* ***cuando*** *se necesiten.*

Como se muestra en la Figura 18.6, la integración primero-en-profundidad integra todos los módulos de un camino de control principal de la estructura. La selección del camino principal es, de alguna manera, arbitraria y dependerá de las características específicas de la aplicación. Por ejemplo, si se elige el camino de la izquierda, se integrarán primero los módulos M1, M2 y M5. **A** continuación, se integrará M8 o M6 (si es necesario para un funcionamiento adecuado de M2). Acto seguido se construyen los caminos de control central y derecho. La integración primero-en-anchura incorpora todos los módulos directamente subordinados a cada nivel, moviéndose por la estructura de forma horizontal. Según la figura, los primeros módulos que se integran son M2, M3 y **M4.** A continuación, sigue el siguiente nivel de control, M5, M6, etc.



¿ Cuáles son los pasos para una integración top-down?

El proceso de integración se realiza en una serie de cinco pasos:

[3] Las estrategias de integración para sistemas orientados a objetos se tratan en el Capitulo 23.

1. Se usa el módulo de control principal como controlador de la prueba, disponiendo de resguardos para todos los módulos directamente subordinados al módulo de control principal.
2. Dependiendo del enfoque de integración elegido (es decir, primero-en-profundidad o primero-en-anchura) se van sustituyendo uno a uno los resguardos subordinados por los módulos reales.
3. Se llevan a cabo pruebas cada vez que se integra un nuevo módulo.
4. Tras terminar cada conjunto de pruebas, se reemplaza otro resguardo con el módulo real.
5. *Se* hace la prueba de regresión (Sección **18.4.3)** para asegurarse de que no se han introducido errores nuevos.

El proceso continúa desde el paso 2 hasta que se haya construido la estructura del programa entero.

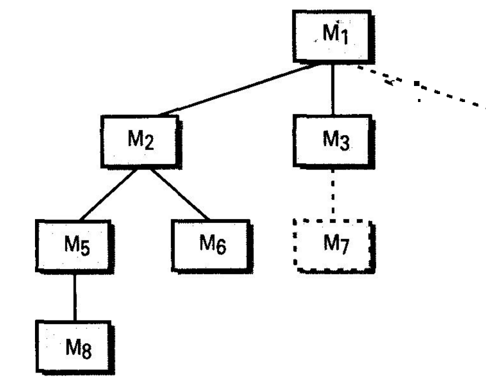


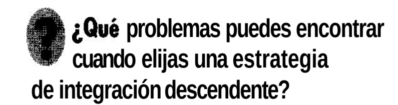
FIGURA 18.6. Integración descendente,

La estrategia de integración descendente verifica los puntos de decisión o de control principales al principio del proceso de prueba. En una estructura de programa bien fabricada, la toma de decisiones se da en los niveles superiores de la jerarquía y, por tanto, se encuentran antes. Si existen problemas generales de control, es esencial reconocerlos cuanto antes. Si se selecciona la integración primero en profundidad, se puede ir implementando y demostrando las funciones completas del software. Por ejemplo, considere una estructura clásica de transacción (Capítulo **14)** en la que se requiere una compleja serie de entradas interactivas, obtenidas y validadas por medio de un camino de entrada. Ese camino de entrada puede ser integrado en forma descendente. Así, se puede demostrar todo el proceso de entradas (para posteriores operaciones de transacción) antes de que se integren otros elementos de la estructura. La demostración anticipada de las posibilidades funcionales es un generador de confianza tanto para el desanollador como para el cliente.

Referencia cruzada

La fabricación es importante *para* ciertos estilos de *arquitectura*. Para **más** detalles ver el Capítulo 14..

La estrategia descendente parece relativamente fácil, pero, en la práctica, pueden surgir algunos problemas logísticos. El más común de estos problemas se da cuando se requiere un proceso de los niveles más bajos de la jerarquía para poder probar adecuadamente los niveles superiores. Al principio de la prueba descendente, los módulos de bajo nivel se reemplazan por ***resguardos;*** por tanto, no pueden fluir datos significativos hacia arriba por la estructura del programa. El responsable de la prueba tiene tres opciones: **(1)** retrasar muchas de las pruebas hasta que los resguardos sean reemplazados por los módulos reales; (2) desarrollar resguardos que realicen funciones limitadas que simulen los módulos reales; o *(3)* integrar el software desde el fondo de la jerarquía hacia arriba.


¿Qué problemas puedes encontrar cuando elijas una estrategia de integración descendente?

El primer enfoque (retrasar pruebas hasta reemplazar los resguardos por los módulos reales) hace que perdamos cierto control sobre la correspondencia de ciertas pruebas específicas con la incorporación de determinados módulos. Esto puede dificultar la determinación de las causas del error y tiende a violar la naturaleza altamente restrictiva del enfoque descendente. El segundo enfoque es factible pero puede llevar a un significativo incremento del esfuerzo a medida que los resguardos se hagan más complejos. El tercer enfoque, denominado prueba ascendente, se estudia en la siguiente sección.

18.4.2. Integración ascendente

La prueba de la integración ascendente, como su nombre indica, empieza la construcción y la prueba con los *módulos atómicos* (es decir, módulos de los niveles más bajos de la estructura del programa). Dado que los módulos se integran de abajo hacia arriba, el proceso requerido de los módulos subordinados siempre está disponible y se elimina la necesidad de resguardos.


¿Cuáles son los pasos para una integración ascendente?

Se puede implementar una estrategia de integración ascendente mediante los siguientes pasos:

1. Se combinan los módulos de bajo nivel en ***grupos*** (a veces denominados construcciones) que realicen una subfunción específica del software.
2. Se escribe un controlador (un programa de control de la prueba) para coordinar la entrada y la salida de los casos de prueba.
3. Se prueba el grupo.
4. Se eliminan los controladores y ***se*** combinan los grupos moviéndose hacia arriba por la estructura del programa.

La integración sigue el esquema ilustrado en la Figura 18.7. Se combinan los módulos para formar los grupos 1, 2 y **3.** Cada uno de los grupos se somete a prueba

mediante un controlador (mostrado como un bloque punteado). Los módulos de los grupos 1 y 2 son subordinados de M_a. Los controladores D_1 y D_2 se eliminan y los grupos interaccionan directamente con M_a. De forma similar, se elimina el controlador D_3 del grupo **3** antes de la integración con el módulo M_b. Tanto M_a como **M_b** se integrarán finalmente con el módulo M_c y así sucesivamente.



La integración ascendente elimina la necesidad de resguardos complejos.

A medida que la integración progresa hacia arriba, disminuye la necesidad de controladores de prueba diferentes. De hecho, si los dos niveles superiores de la estructura del programa se integran de forma descendente, se puede reducir sustancialmente el número de controladores y se simplifica enormemente la integración de grupos.

18.4.3. Prueba de regresión

Cada vez que se añade un nuevo módulo como parte de una prueba de integración, el software cambia. Se establecen nuevos caminos de flujo de datos, pueden ocurrir nuevas E/S y se invoca una nueva lógica de control. Estos cambios pueden causar problemas con funciones que antes trabajaban perfectamente. En el contexto de una estrategia de prueba de integración, la *prueba de regresión* es volver a ejecutar un subconjunto de pruebas que se han llevado a cabo anteriormente para asegurarse de que los cambios no han propagado efectos colaterales no deseados.

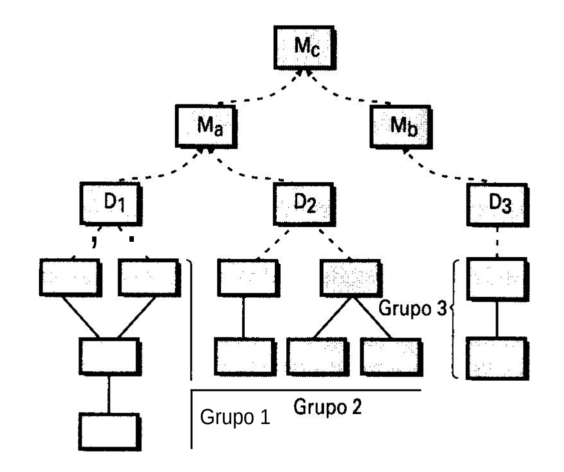


FIGURA 18.7. Integración ascendente.

En un contexto más amplio, las pruebas con éxito (de cualquier tipo) dan como resultado el descubrimiento de errores, y los errores hay que corregirlos. Cuando se corrige el software, se cambia algún aspecto de la configuración del software (el programa, su documentación o los datos que lo soportan). La prueba de regresión es la actividad que ayuda a asegurar que los cambios (debidos a las pruebas o por otros motivos) no introducen un comportamiento no deseado o errores adicionales.



La prueba de regresión es una estrategia importante para reducir «efectos colaterales». Se deben ejecutar pruebas de regresión cada vez que se realice un cambio importante en el software (incluyendo la integración de nuevos módulos).

La prueba de regresión se puede hacer manualmente, volviendo a realizar un subconjunto de todos los casos de prueba o utilizando herramientas automáticas de reproducción de captura. Las *herramientas de reproducción **de** captura* permiten al ingeniero del software capturar casos de prueba y los resultados para la subsiguiente reproducción y comparación. El conjunto de pruebas de regresión (el subconjunto de pruebas a realizar) contiene tres clases diferentes de casos de prueba:

- una muestra representativa de pruebas que ejercite todas las funciones del software;
- pruebas adicionales que se centran en las funciones del software que se van a ver probablemente afectadas por el cambio;
- pruebas que se centran en los componentes del software que han cambiado.

A medida que progresa la prueba de integración, el número de pruebas de regresión puede crecer demasiado. Por tanto, el conjunto de pruebas de regresión debería diseñarse para incluir sólo aquellas pruebas que traten una o más clases de errores en cada una de las funciones principales del programa. No es práctico ni eficiente volver a ejecutar cada prueba de cada función del programa después de un cambio.

18.4.4. Prueba de humo

La prueba de humo es un método de prueba de integración que es comúnmente utilizada cuando se ha desarrollado un producto software «empaquetado». **Es** diseñado como un mecanismo para proyectos críticos por tiempo, permitiendo que el equipo de software valore su proyecto sobre una base sólida. En esencia, la prueba de humo comprende las siguientes actividades:

1. Los componentes software que han sido traducidos a código se integran en una «construcción». Una construcción incluye ficheros de datos, librerías, módulos reutilizables y componentes de ingeniería que se requieren para implementar una o más funciones del producto.
2. Se diseña una serie de pruebas para descrubir errores que impiden a la construcción realizar su fun-

ción adecuadamente. El objetivo será descubrir errores «bloqueantes» que tengan la mayor probabilidad de impedir al proyecto de software el cumplimiento de su planificación.

3. Es habitual en la prueba de humo que la construcción se integre con otras construcciones y que se aplica una prueba de humo al producto completo (en su forma actual). La integración puede hacerse bien de forma descendente *(top-down)* o ascendente *(bottom-up)*.



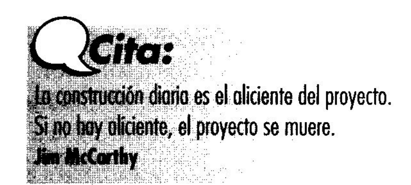
La prueba de humo se caracteriza por una estrategia de integración continua. El software es reconfigurado (con la incorporación de nuevos componentes) y utilizado continuamente.

La frecuencia continua de la prueba completa del producto puede sorprender a algunos lectores. En cualquier caso, las frecuentes pruebas dan a gestores y profesionales una valoración realista de la evolución de las pruebas de integración. McConnell [MCO96] describe la prueba de humo de la siguiente forma:

> La prueba de humo ejercita el sistema entero de principio a fin. No ha de ser exhaustiva, pero será capaz de descubrir importantes problemas. La prueba de humo será suficiente si verificamos de forma completa la construcción y podemos asumir que es suficientemente estable para ser probado con más profundidad.

La prueba de humo facilita una serie de beneficios cuando se aplica sobre proyectos de ingeniería del software complejos y críticos por su duración:

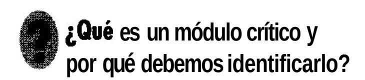
- *Se minimizan los riesgos de integración.* Dado que las pruebas de humo son realizadas frecuentemente, incompatibilidades y otros errores bloqueantes son descubiertos rápidamente, por eso se reduce la posibilidad de impactos importantes en la planificación por errores sin descubrir.
- *Se perfecciona la calidad del producto final.* Dado que la prueba de humo es un método orientado a la construcción (integración), es probable que descubra errores funcionales, además de defectos de diseño a nivel de componente y de arquitectura. Si estos defectos se corrigen rápidamente, el resultado será un producto de gran calidad.



- *Se simplifican el diagnóstico y la corrección de errores.* Al igual que todos los enfoques de prueba de integración, es probable que los errores sin descubrir durante la prueba de humo se asocien a «nuevos incrementos de software» - esto es, el software que se acaba de añadir a la construcción es una posible causa de un error que se acaba de descubrir—.
- *El progreso es fácil de observar.* Cada día que pasa, se integra más software y se demuestra que funciona. Esto mejora la moral del equipo y da una indicación a los gestores del progreso que se está realizando.

18.4.5. Comentarios sobre la prueba de integración

Ha habido muchos estudios (por ejemplo, [BEI84]) sobre las ventajas y desventajas de la prueba de integración ascendente frente a la descendente. En general, las ventajas de una estrategia tienden a convertirse en desventajas para la otra estrategia. La principal desventaja del enfoque descendente es la necesidad de resguardos y las dificultades de prueba que pueden estar asociados con ellos. Los problemas asociados con los resguardos pueden quedar compensados por la ventaja de poder probar de antemano las principales funciones de control. La principal desventaja de la integración ascendente es que «el programa como entidad no existe hasta que se ha añadido el último módulo» [MYE79]. Este inconveniente se resuelve con la mayor facilidad de diseño de casos de prueba y con la falta de resguardos.



La selección de una estrategia de integración depende de las características del software y, a veces, de la planificación del proyecto. En general, el mejor compromiso puede ser un enfoque combinado (a veces denominado *prueba sandwich)* que use la descendente para los niveles superiores de la estructura del programa, junto con la ascendente para los niveles subordinados.

A medida que progresa la prueba de integración, el responsable de las pruebas debe identificar los módulos críticos. Un módulo crítico es aquel que tiene una o más de las siguientes características: (1) está dirigido a varios requisitos del software; (2) tiene un mayor nivel de control (está relativamente alto en la estructura del programa); (3) es complejo o propenso a errores (se puede usar la complejidad ciclomática como indicador); o (4) tiene unos requisitos de rendimiento muy definidos. Los módulos críticos deben probarse lo antes posible. Además, las pruebas de regresión se deben centrar en el funcionamiento de los módulos críticos.

18.5 PRUEBA DE VALIDACIÓN

Tras la culminación de la prueba de integración, el software está completamente ensamblado como un paquete, se han encontrado y corregido los errores de interfaz y puede comenzar una serie final de pruebas del software: la *prueba* de validación. La validación puede definirse de muchas formas, pero una simple (aunque vulgar) definición es que la validación se consigue cuando el software funciona de acuerdo con las expectativas razonables del cliente. En este punto, un desarrollador de software estricto podría protestar: «¿Qué o quién es el árbitro de las expectativas razonables?»



CLAVE

Como en otras etapas de la prueba, la validación permite descubrir errores, pero su enfoque está en el nivel de requisitos —sobre cosas que son necesarias para el usuario final—.

Las expectativas razonables están definidas en la Especificación de Requisitos del Software —un documento (Capítulo 12) que describe todos los atributos del software visibles para el usuario. La especificación contiene una sección denominada —. «Criterios de validación». La información contenida en esa sección forma la base del enfoque a la prueba de validación.

18.5.1. Criterios de la prueba de validación

La validación del software se consigue mediante una serie de pruebas de caja negra que demuestran la conformidad con los requisitos. Un plan de prueba traza la clase de pruebas que se han de llevar a cabo, y un procedimiento de prueba define los casos de prueba específicos en un intento por descubrir errores de acuerdo con los requisitos. Tanto el plan como el procedimiento estarán diseñados para asegurar que se satisfacen todos los requisitos funcionales, que se alcanzan todos los requisitos de rendimiento, que la documentación es correcta e inteligible y que se alcanzan otros requisitos (por ejemplo, portabilidad, compatibilidad, recuperación de errores, facilidad de mantenimiento).

Una vez que se procede con cada caso de prueba de validación, puede darse una de las dos condiciones siguientes: (1) las características de funcionamiento o de rendimiento están de acuerdo con las especificaciones y son aceptables; o (2) se descubre una desviación de las especificaciones y se crea una lista de deficiencias. Las desviaciones o errores descubiertos en esta fase del proyecto raramente se pueden corregir antes de la terminación planificada. A menudo es necesario negociar con el cliente un método para resolver las deficiencias.

18.5.2. Revisión de la configuración

Un elemento importante del proceso de validación es la revisión de *la* configuración. La intención de la revisión es asegurarse de que todos los elementos de la configuración del software se han desarrollado apropiadamente, se han catalogado y están suficientemente detallados para soportar la fase de mantenimiento durante el ciclo de vida del software. La revisión de la configuración, a veces denominada *auditoría*, se ha estudiado con más detalle en el Capítulo 9.

18.5.3. Pruebas alfa y beta

Es virtualmente imposible que un desarrollador de software pueda prever cómo utilizará el usuario realmente el programa. Se pueden malinterpretar las instrucciones de uso, se pueden utilizar habitualmente extrañas combinaciones de datos, y una salida que puede parecer clara para el responsable de las pruebas y puede ser ininteligible para el usuario.

Cuando se construye software a medida para un cliente, se llevan a cabo una serie de pruebas de aceptación para permitir que el cliente valide todos los requisitos. Las realiza el usuario final en lugar del responsable del desarrollo del sistema, una prueba de aceptación puede ir desde un informal «paso de prueba» hasta la ejecución sistemática de una serie de pruebas bien planificadas. De hecho, la prueba de aceptación puede tener lugar a lo largo de semanas o meses, descubriendo así errores acumulados que pueden ir degradando el sistema.

Si el software se desarrolla como un producto que va a ser usado por muchos clientes, no es práctico realizar pruebas de aceptación formales para cada uno de ellos. La mayoría de los desarrolladores de productos de software llevan a cabo un proceso denominado prueba alfa y beta para descubrir errores que parezca que sólo el usuario final puede descubrir.

La prueba *alfa* se lleva a cabo, por un cliente, en el lugar de desarrollo. Se usa el software de forma natural con el desarrollador como observador del usuario y registrando los errores y los problemas de uso. Las pruebas alfa se llevan a cabo en un entorno controlado.

La *prueba beta* se lleva a cabo por los usuarios finales del software en los lugares de trabajo de los clientes. **A** diferencia de la prueba alfa, el desarrollador no está presente normalmente. **Así,** la prueba beta es una aplicación «en vivo» del software en un entorno que no puede ser controlado por el desarrollador. El cliente registra todos los problemas (reales o imaginarios) que encuentra durante la prueba beta e informa a intervalos regulares al desarrollador. Como resultado de los problemas informados durante la prueba beta, el desarrollador del software lleva a cabo modificaciones y así prepara una versión del producto de software para toda la clase de clientes.

18.6 PRUEBA DEL SISTEMA

Al comienzo de este libro, pusimos énfasis en el hecho de que el software es sólo un elemento de un sistema mayor basado en computadora. Finalmente, el software es incorporado a otros elementos del sistema (por ejemplo, nuevo hardware, información) y realizan una serie de pruebas de integración del sistema y de validación. Estas pruebas caen fuera del ámbito del proceso de ingeniería del software y no las realiza únicamente el desarrollador del software. Sin embargo, los pasos dados durante el diseño del software y durante la prueba pueden mejorar enormemente la probabilidad de éxito en la integración del software en el sistema.



Semejante a la muerte y a los impuestos, la prueba es desagradable e inevitable.
Ed Yourdon

Un problema típico de la prueba del sistema es la ((delegaciónde culpabilidad». Esto ocurre cuando se descubre un error y cada uno de los creadores de cada elemento del sistema echa la culpa del problema a los otros. En vez de verse envuelto en esta absurda situación, el ingeniero del software debe anticiparse a los posibles problemas de interacción y: (1) diseñar caminos de manejo de errores que prueben toda la información procedente de otros elementos del sistema; (2) llevar a cabo una serie de pruebas que simulen la presencia de datos en mal estado o de otros posibles errores en la interfaz del software; (3) registrar los resultados de las pruebas como «evidencia» en el caso de que se le señale con el dedo; (4) participar en la planificación y el diseño de pruebas del sistema para asegurarse de que el software se prueba de forma adecuada.

La prueba del sistema, realmente, está constituida por una serie de pruebas diferentes cuyo propósito primordial es ejercitar profundamente el sistema basado en computadora. Aunque cada prueba tiene un propósito diferente, todas trabajan para verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas. En las siguientes secciones examinamos los tipos de pruebas del sistema [BEI84] valiosos para sistemas basados en software.

18.6.1. Prueba de recuperación

Muchos sistemas basados en computadora deben recuperarse de los fallos y continuar el proceso en un tiempo previamente especificado. En algunos casos, un sistema debe ser tolerante con los fallos; es decir, los fallos del proceso no deben hacer que cese el funcionamiento de todo el sistema.



Amplia información sobre la prueba del software y su relación con las necesidades de calidad pueden obtenerse en **www.stqe.net**

En otros casos, se debe corregir un fallo del sistema en un determinado periodo de tiempo para que no se produzca un serio daño económico.

La *prueba de recuperación* es una prueba del sistema que fuerza el fallo del software de muchas formas y verifica que la recuperación se lleva a cabo apropiadamente. Si la recuperación es automática (llevada a cabo por el propio sistema) hay'que evaluar la corrección de la inicialización, de los mecanismos de recuperación del estado del sistema, de la recuperación de datos y del proceso de rearranque. Si la recuperación requiere la intervención humana, hay que evaluar los tiempos medios de reparación (TMR) para determinar si están dentro de unos límites aceptables.

18.6.2. Prueba de seguridad

Cualquier sistema basado en computadora que maneje información sensible o lleve a cabo acciones que puedan perjudicar (o beneficiar) impropiamente a las personas es un posible objetivo para entradas impropias o ilegales al sistema. Este acceso al sistema incluye un amplio rango de actividades: *«piratas informáticos»* que intentan entrar en los sistemas por deporte, empleados disgustados que intentan penetrar por venganza e individuos deshonestos que intentan penetrar para obtener ganancias personales ilícitas.

La *prueba de seguridad* intenta verificar que los mecanismos de protección incorporados en el sistema lo protegerán, de hecho, de accesos impropios. Para citar a Beizer [BEI84]: «Por supuesto, la seguridad del sistema debe ser probada en su invulnerabilidad frente a un ataque frontal, pero también debe probarse en su invulnerabilidad a ataques por los flancos o por la retaguardia.»

Durante la prueba de seguridad, el responsable de la prueba desempeña el papel de un individuo que desea entrar en el sistema. ¡Todo vale! Debe intentar conseguir las claves de acceso por cualquier medio, puede atacar al sistema con software a medida, diseñado para romper cualquier defensa que se haya construido, debe bloquear el sistema, negando así el servicio a otras personas, debe producir a propósito errores del sistema, intentando acceder durante la recuperación o debe curiosear en los datos sin protección, intentando encontrar la clave de acceso al sistema, etc.

Con tiempo y recursos suficientes, una buena prueba de seguridad terminará por acceder al sistema. **El**

papel del diseñador del sistema es hacer que el coste de la entrada ilegal sea mayor que el valor de la información obtenida.



Si has intentado encontrar verdaderamente fallos en el sistema y nunca has sometido tu software a una prueba real de resistencia, entonces has perdido mucho el tiempo.
Boris Beizer

18.6.3. Prueba de resistencia (Stress)

Durante los pasos de prueba anteriores, las técnicas de caja blanca y de caja negra daban como resultado la evaluación del funcionamiento y del rendimiento normales del programa. Las pruebas de resistencia están diseñadas para enfrentar a los programas con situaciones anormales. En esencia, el sujeto que realiza la prueba de resistencia se pregunta: «¿A qué potencia puedo ponerlo a funcionar antes de que falle?»

La *prueba de resistencia* ejecuta un sistema de forma que demande recursos en cantidad, frecuencia o volúmenes anormales. Por ejemplo: (1) diseñar pruebas especiales que generen diez interrupciones por segundo, cuando las normales son una o dos; (2) incrementar las frecuencias de datos de entrada en un orden de magnitud con el fin de comprobar cómo responden las funciones de entrada; (3) ejecutar casos de prueba que requieran el máximo de memoria o de otros recursos; (4) diseñar casos de prueba que puedan dar problemas en un sistema operativo virtual o (5) diseñar casos de prueba que produzcan excesivas búsquedas de datos residentes en disco. Esencialmente, el responsable de la prueba intenta romper el programa.

Una variante de la prueba de resistencia es una técnica denominada prueba de *sensibilidad*. En algunas situaciones (la más común se da con algoritmos matemáticos), un rango de datos muy pequeño dentro de los límites de una entrada válida para un programa puede producir un proceso exagerado e incluso erróneo o una profunda degradación del rendimiento. Esta situación es análoga a una singularidad en una función matemática. La prueba de sensibilidad intenta descubrir combinaciones de datos dentro de una clase de entrada válida que pueda producir inestabilidad o un proceso incorrecto.

18.6.4. Prueba de rendimiento

Para sistemas de tiempo real y sistemas empotrados, es inaceptable el software que proporciona las funciones requeridas pero no se ajusta a los requisitos de rendimiento. La *prueba de rendimiento* está diseñada para probar el rendimiento del software en tiempo de ejecución dentro del contexto de un sistema integrado. La prueba de rendimiento se da durante todos los pasos del proceso de la prueba. Incluso al nivel de unidad, se debe asegurar el rendimiento de los módulos individuales a medida que se llevan a cabo las pruebas de caja blanca. Sin embargo, hasta que no están completamente integrados todos los elementos del sistema no se puede asegurar realmente el rendimiento del sistema.

Las pruebas de rendimiento, a menudo, van emparejadas con las pruebas de resistencia y, frecuentemente, requieren instrumentación tanto de software como de hardware. Es decir, muchas veces es necesario medir la utilización de recursos (por ejemplo, ciclos de procesador), de un modo exacto. La instrumentación externa puede monitorizar los intervalos de ejecución, los sucesos ocurridos (por ejemplo, interrupciones) y muestras de los estados de la máquina en un funcionamiento normal. Instrumentando un sistema, el encargado de la prueba puede descubrir situaciones que lleven a degradaciones y posibles fallos del sistema.

18.7 EL ARTE DE LA DEPURACIÓN

La prueba del software es un proceso que puede planificarse y especificarse sistemáticamente. Se puede llevar a cabo el diseño de casos de prueba, se puede definir una estrategia y se pueden evaluar los resultados en comparación con las expectativas prescritas.

La *depuración* ocurre como consecuencia de una prueba efectiva. Es decir, cuando un caso de prueba descubre un error, la depuración es el proceso que provoca la eliminación del error. Aunque la depuración puede y debe ser un proceso ordenado, sigue teniendo mucho de arte. Un ingeniero del software, al evaluar los resultados de una prueba, se encuentra frecuentemente con una indicación «sintomática» de un problema en el software. Es decir, la manifestación externa de un error, y la causa interna del error pueden no estar relacionados de una forma obvia. El proceso mental, apenas comprendido, que conecta un síntoma con una causa es la depuración.



BugNet facilita información sobre problemas de seguridad y fallos en software basado en PC y proporciona una información útil sobre temas de depuración:
www.bugnet.com

18.7.1. El Proceso de depuración

La depuración no es una prueba, pero siempre ocurre como consecuencia de la prueba[4]. Como se muestra en la Figura 18.8, el proceso de depuración comienza con la ejecución de un caso de prueba. Se evalúan los resultados y aparece una falta de correspondencia entre los esperados y los encontrados realmente. En muchos casos, los datos que no concuerdan son un síntoma de una causa subyacente que todavía permanece oculta. El proceso de depuración intenta hacer corresponder el sistema con una causa, llevando así a la corrección del error.

El proceso de depuración siempre tiene uno de los dos resultados siguientes: (1) se encuentra la causa, se corrige y se elimina; o (2) no se encuentra la causa. En este último caso, la persona que realiza la depuración debe sospechar la causa, diseñar un caso de prueba que ayude a confirmar sus sospechas y el trabajo vuelve hacia atrás a la corrección del error de una forma iterativa.

¿Por qué es tan difícil la depuración? Todo parece indicar que la respuesta tiene más que ver con la psicología humana (véase la siguiente sección) que con la tecnología del software. Sin embargo, varias características de los errores nos dan algunas pistas:

1. El síntoma y la causa pueden ser geográficamente remotos entre sí. Es decir, el síntoma puede aparecer en una parte del programa, mientras que la causa está localizada en otra parte muy alejada. Las estructuras de programa fuertemente acopladas (Capítulo 13) resaltan esta situación.
2. El síntoma puede desaparecer (temporalmente) al corregir otro error.
3. El síntoma puede realmente estar producido por algo que no es un error (por ejemplo, inexactitud en los redondeos).
4. El síntoma puede estar causado por un error humano que no sea fácilmente detectado.
5. El síntoma puede ser el resultado de problemas de temporización en vez de problemas de proceso.
6. Puede ser difícil reproducir exactamente las condiciones de entrada (por ejemplo, una aplicación de tiempo real en la que el orden de la entrada no está determinado).



La variedad que dentro de los programas de computador debe diagnosticarse [para depuración] es probablemente mayor que la variedad en otros ejemplos de sistemas que son regularmente diagnosticados.
John Gould

7. El síntoma puede aparecer de forma intermitente. Esto es particularmente comente en sistemas empotrados que acoplan el hardware y el software de manera confusa.
8. El síntoma puede ser debido a causas que se distribuyen por una serie de tareas ejecutándose en diferentes procesadores [CHE90].

Durante la depuración encontramos errores que van desde lo ligeramente inesperado (por ejemplo, un formato de salida incorrecto) hasta lo catastrófico (por ejemplo, el sistema falla, produciéndose serios daños económicos o físicos). A medida que las consecuencias de un error aumentan, crece la presión por encontrar su causa. A menudo la presión fuerza a un ingeniero del software a corregir un error introduciendo dos más.

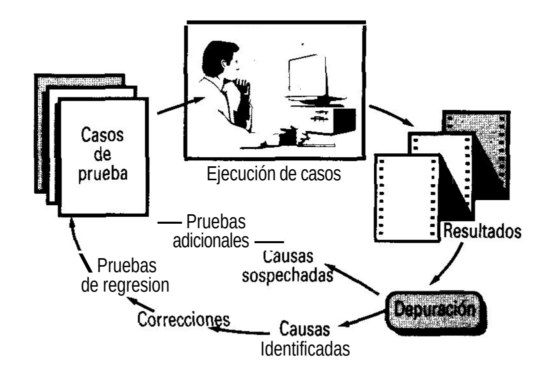


FIGURA 18.8. El proceso de depuración.

18.7.2. Consideraciones psicológicas

Desafortunadamente, todo parece indicar que la habilidad en la depuración es un rasgo innato del ser humano. A ciertas personas se les da bien y a otras no. Aunque las manifestaciones experimentales de la depuración están abiertas a muchas interpretaciones, se han detectado grandes variaciones en la destreza para la depuración de distintos programadores con el mismo bagaje de formación y de experiencia.

Hablando de los aspectos humanos de la depuración, Shneiderman [SHN80] manifiesta:

> La depuración es una de las partes más frustrantes de la programación. Contiene elementos de resolución de problemas o de rompecabezas, junto con el desagradable reconocimiento de que se ha cometido un error. La enorme ansiedad y la **no** inclinación a aceptar la posibilidad de cometer errores hace que la tarea sea extremadamente difícil. Afortunadamente, también se da un gran alivio y disminuye la tensión cuando el error **es** finalmente... corregido.

Aunque puede resultar difícil «aprender» a depurar, se pueden proponer varios enfoques del problema. En la siguiente sección los examinamos.

[4] Al decir esta frase, tomamos el punto de vista más amplio que se puede tener de la prueba. No sólo ha de llevar a cabo la prueba el equipo de desarrollo antes de entregar el software, sino que el cliente/usuario prueba el software cada vez que lo usa.

18.7.3. Enfoques de la depuración

Independientemente del enfoque que se utilice, la depuración tiene un objetivo primordial: encontrar y corregir la causa de un error en el software. El objetivo se consigue mediante una combinación de una evaluación sistemática, de intuición y de suerte. Bradley [BRA85] describe el enfoque de la depuración de la siguiente forma:

> La depuración es una aplicación directa del método científico desarrollado hace 2.500 años. La base de la depuración es la localización de la fuente del problema [la causa] mediante partición binaria, manejando hipótesis que predigan nuevos valores a examinar.
>
> Tomemos un sencillo ejemplo que no tiene que ver con el software: en mi casa no funciona una lámpara. Si no funciona nada en la casa, la causa debe estar en el circuito principal de fusibles o fuera de la casa; miro fuera para ver si hay un apagón en todo el vecindario. Conecto la sospechosa lámpara a un enchufe que funcione y un aparato que funcione en el circuito sospechoso. **Así** se sigue la secuencia de hipótesis y de pruebas.

En general, existen tres enfoques que se pueden proponer para la depuración [MYE79]:

1. Fuerza bruta
2. Vuelta atrás
3. Eliminación de causas

La categoría de depuración por la *fuerza bruta* es probablemente la más común y menos eficiente a la hora de aislar la causa del error en el software. Aplicamos los métodos de depuración por fuerza bruta cuando todo lo demás falla. Mediante una filosofía de «dejar que la computadora encuentre el error», se hacen volcados de memoria, trazas de ejecución y se cargan multitud de sentencias *Mostrar* en el programa. Esperamos que en algún lugar de la gran cantidad de información generada encontremos alguna pista que nos lleve a la causa de un error. Aunque la gran cantidad de información producida nos puede llevar finalmente al éxito, lo más frecuente es que se desperdicie tiempo y esfuerzo. ¡Primero se debe usar la inteligencia!

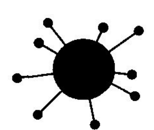


fíjate un tiempo limitado, por ejemplo dos horas, en relación a la cantidad de tiempo que tu inviertes para depurar un problema de tu incumbencia. Después qué, ¡pide ayuda!

La *vuelta atrás* es un enfoque más normal para la depuración, que se puede usar con éxito para pequeños programas. Partiendo del lugar donde se descubre el síntoma, ***se*** recorre hacia atrás (manualmente) el código fuente hasta que se llega a la posición de error. Desgraciadamente, a medida que aumenta el número de líneas del código, el número de posibles caminos de vuelta se hace difícilmente manejable.

El tercer enfoque para la depuración —*eliminación de causas*— se manifiesta mediante inducción o deducción e introduce el concepto de partición binaria. Los datos relacionados con la ocurrencia del error se organizan para aislar las posibles causas. Se llega a una «hipótesis de causa» y se usan los datos anteriores para probar o revocar la hipótesis. Alternativamente, se desarrolla una lista de todas las posibles causas y se llevan a cabo pruebas para eliminar cada una. Si alguna prueba inicial indica que determinada hipótesis de causa en particular parece prometedora, se refinan los datos con el fin de intentar aislar el error.

Cada uno de los enfoques anteriores puede complementarse con herramientas de depuración. Podemos usar una gran cantidad de compiladores de depuración, ayudas dinámicas para la depuración («trazadores»), generadores automáticos de casos de prueba, volcados de memoria y mapas de referencias cruzadas. Sin embargo, las herramientas no son un sustituto de la evaluación cuidadosa basada en un completo documento del diseño del software y un código fuente claro.



Herramientas CASE
Pruebo y Depuración.

Cualquier discusión sobre los enfoques para la depuración y sus herramientas no estaría completa sin mencionar un poderoso aliado: ¡otras personas! Cualquiera de nosotros podrá recordar haber estado dando vueltas en la cabeza durante horas o días a un error persistente. Desesperados, le explicamos el problema a un colega con el que damos por casualidad y le mostramos el listado. Instantáneamente (parece), se descubre la causa del error. Nuestro colega se aleja sonriendo ladinamente. Un punto de vista fresco, no embotado por horas de frustración, puede hacer maravillas. Una máxima final para la depuración puede ser: «¡Cuando todo lo demás falle, pide ayuda!»

Una vez que se ha encontrado un error, hay que corregirlo. Pero como ya hemos podido observar, la corrección de un error puede introducir otros errores y hacer más mal que bien.

? Cuando corrijo un error, ¿qué cuestiones debo preguntarme a mi mismo?

Van Vleck [VAN89] sugiere tres preguntas sencillas que debería preguntarse todo ingeniero del software antes de hacer la «corrección» que elimine la causa del error:

1 *¿Se repite la causa del error en otra parte del programa?* En muchas situaciones, el defecto de un programa está producido por un patrón de lógica erróneo que se puede repetir en cualquier lugar. La consideración explícita del patrón lógico puede terminar en el descubrimiento de otros errores.

2. *¿Cuál es el «siguiente error» que se podrá presentar a raíz de la corrección que hoy voy a realizar?* Antes de hacer la corrección, se debe evaluar el código fuente (o mejor, el diseño) para determinar el emparejamiento de la lógica y las estructuras de datos. Si la corrección se realiza en una sección del programa altamente acoplada, se debe tener cuidado al realizar cualquier cambio.

3. *¿Qué podríamos haber hecho para prevenir este error la primera vez?* Esta pregunta es el primer paso para establecer un método estadístico de garantía de calidad del software (Capítulo **8**). Si corregimos tanto el proceso como el producto, se eliminará el error del programa actual y se puede eliminar de todos los futuros programas.

RESUMEN

La prueba del software contabiliza el mayor porcentaje del esfuerzo técnico del proceso de desarrollo de software. Todavía estamos comenzando a comprender las sutilezas de la planificación sistemática de la prueba, de su ejecución y de su control.

El objetivo de la prueba de software es descubrir errores. Para conseguir este objetivo, se planifica y se ejecutan una serie de pasos; pruebas de unidad, de integración, de validación y del sistema. Las pruebas de unidad y de integración se centran en la verificación funcional de cada módulo y en la incorporación de los módulos en una estructura de programa. La prueba de validación demuestra el seguimiento de los requisitos del software y la prueba del sistema valida el software una vez que se ha incorporado en un sistema superior.

Cada paso de prueba se lleva a cabo mediante una serie de técnicas sistemáticas de prueba que ayudan en el diseño de casos de prueba. Con cada paso de prueba se amplía el nivel de abstracción con el que se considera el software.

A diferencia de la prueba (una actividad sistemática y planificada), la depuración se puede considerar un arte. **A** partir de una indicación sintomática de un problema, la actividad de la depuración debe rastrear la causa del error. De entre los recursos disponibles durante la depuración, el más valioso puede ser el apoyo de otros ingenieros de software.

El requisito de que el software sea cada vez de mayor calidad exige un planteamiento más sistemático de la prueba. Citando a Dunn y Ullman [DUN82]:

> Lo que se requiere es una estrategia global, que se extienda por el espacio estratégico de la prueba, **tan** deliberada en su metodología como lo fue el desarrollo sistemático en el que se basaron el análisis, el diseño y la codificación.

En este capítulo hemos examinado el espacio estratégico de la prueba, considerando los pasos que tienen la mayor probabilidad de conseguir el fin Último de la prueba: encontrar y subsanar los defectos de una manera ordenada **y** efectiva.

REFERENCIAS

[BEI84] Beizer, B., *Software System Testing and Quality Assurance,* Van Nostrand Reinhold, 1984.

[BOE81] Boehm, B., *Software Engineering Economics,* Prentice-Hall, 1981, p. 37.

[BRA85] Bradley, J.H., «The science and Art of Debugging», Computerworld, 19 de Agosto de 1985, pp. 35-38.

[CHE90] Cheung, W. H., J. P. Black y E. Manning, «A Framework for Distributed Debugging», *IEEE Software*, Enero 1990, pp. 106-115.

[DUN82] Dunn, R., y R. Ullman, *Quality Assurance for Computer Software*, McGraw-Hill, 1982, p. 158.

[GIL95] Gilb, T., «What We Fail To Do In Our Current Testing Culture», *Testing Techniques Newsletter,* (edición en-línea, ttn@soft.com), Software Research, Inc, San Francisco, Enero 199.5.

[MCO96] McConnell, S., «Best Practices: Daily Build and Smoke Test», IEEE Software, vol. 13, n.º 4, Julio 1996, pp. 143-144.

[MILL77] Miller, E., «The Philosophy of Testing», *Program Testing Techniques,* IEEE Computer Society Press, 1977, pp. 1-3.

[MUS89] Musa, J. D., y Ackerman, A. F., «Quantifying Software Validation: When to Stop Testing?», *IEEE Software,* Mayo 1989, pp. 19-27.

[MYE79] Myers, G., *The Art of Software Tests,* Wiley, 1979.

[SHO83] Shooman, **M.** L., *Software Engineering,* Mcgraw-Hill, 1983.

[SHN80] Shneiderman, B., *Software Psychology,* Winthrop Publishers, 1980, p. 28.

[VAN89] Van Bleck, T., «Three Questions About Each Bug You find», *ACM Sofiare Engineering Notes,* vol. 14, n.º 5, Julio 1989, pp. 62-63.

[WAL89] Wallance, D. R., y R. U. Fujii, «Software Verification and Validation: An Overview», *IEEE Software*, Mayo 1989, pp. 10-17.

[YOU75] Yourdon, E., *Techniques of Progrum Structure and Design*, Prentice-Hall, 1975.

PROBLEMAS Y PUNTOS A CONSIDERAR

18.1. Con sus propias palabras, describa las diferencias entre verificación y validación. ¿Utilizan las dos los métodos de diseño de casos de prueba y las estrategias de prueba?

18.2. Haga una lista de algunos problemas que puedan estar asociados con la creación de un grupo independiente de prueba. ¿Están formados por las mismas personas el GIP y el grupo SQA?

18.3. ¿Es siempre posible desarrollar una estrategia de prueba de software que use la secuencia de pasos de prueba descrita en la Sección 18.1.3? ¿Qué posibles complicaciones pueden surgir para sistemas empotrados?

18.4. Si sólo pudiera seleccionar tres métodos de diseño de casos de prueba para aplicarlos durante la prueba de unidad, ¿cuáles serían y por qué?

18.5. Porqué es difícil de realizar la prueba unitaria a un módulo con alto nivel de integración.

18.6. Desarrolle una estrategia de prueba de integración para cualquiera de los sistemas implementados en los problemas 16.4 a 16.11. Defina las fases de prueba, indique el orden de integración, especifique el software de prueba adicional y justifique el orden de integración. Suponga que todos los módulos han sido probados en unidad y están disponibles. [Nota: puede ser necesario comenzar trabajando un poco con el diseño inicialmente.]

18.7. ¿Cómo puede afectar la planificación del proyecto a la prueba de integración?

18.8. ¿Es posible o incluso deseable la prueba de unidad en cualquier circunstancia? Ponga ejemplos que justifiquen su respuesta.

18.9. ¿Quién debe llevar a cabo la prueba de validación —el desarrollador del software o el usuario—? Justifique su respuesta.

18.10. Desarrolle una estrategia de prueba completa para el sistema *HogarSeguro* descrito anteriormente en este libro. Documéntela en una *Especificación de Prueba.*

18.11. Como proyecto de clase, desarrolle una guía de depuración para su instalación. La guía debe proporcionar consejos orientados al lenguaje y al sistema que se hayan aprendido con las malas experiencias. Comience con un esbozo de los temas que se tengan que revisar por la clase y **su** profesor. Publique la guía para otras personas de su entorno.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Libros orientados a las estrategias de prueba del software son los de Black (*Managing the Testing Process*, Microsoft Press, 1999), Dustin, Rashka and Paul (*Test Process Improvement*: Step-By-Step Guide to Structured Testing, Addison-Wesley, 1999), Perry (*Suwiving the Top Ten Challenges of Software Testing: A People-Oriented Approach,* Dorset House, 1997), and Kit and Finzi (*Software Testing in the Real World: Improving the Process,* Addison-Wesley, 1995).

Kaner, Nguyen y Falk (*Testing Computer Softwa-re,* Wiley, 1999), Hutcheson (*Software Testing Methods and Metrics: The Most Important Tests,* McGraw Hill, 1997), Marick (*The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing,* Prentice Hall, 1995), Jorgensen (*Software Testing: A Crafts-man 's Approach,* CRC Press, 1995) presentan estudios sobre los métodos y estrategias de prueba.

Además, antiguos libros de Evans (*Productive Software Test Management*, Wiley-Interscience, 1984), Hetzel (*The Complete Guide to Software Testing,* QED Information Sciences, 1984), Beizer [BEI84], Ould y Unwin (*Testing in Software Development,* Cambridge University Press, 1986), Marks (*Testing Very Big Systems*, McGraw-Hill, 1992), y Kaner et al. (*Testing Computer Software,* 2.ª ed., Van Nostrand Reinhold, 1993) define los pasos para una estrategia efectiva, proporciona un conjunto de técnicas y directrices y sugiere procedimientos para controlar y hacer un seguimiento a los procesos de prueba. Hutcheson (*Software Testing Methods and Metrics,* McGraw-Hill, 1996) presenta unos métodos y estrategias de prueba pero también proporciona un estudio detallado de cómo se puede usar la medición para conseguir una prueba efectiva.

Un libro de Dunn (*Software Defect Removal,* McGraw-Hill, 1984) contiene unas directrices para la depuración. Beizer [BEI84] presenta una interesante «taxonomía de errores» que puede conducir a unos métodos efectivos para la planificación de pruebas. McConnell (*Code Complete,* Microsoft Press, 1993) presenta unos pragmáticos consejos sobre las pruebas de unidad y de integración así como sobre la depuración.

Una amplia variedad de fuentes de información sobre pruebas del software y elementos relacionados están disponibles en Internet. Una lista actualizada de páginas web sobre conceptos de prueba, métodos y estrategias pueden encontrarse en **http://www.pressman5.com.**

19 MÉTRICAS TÉCNICAS DEL SOFTWARE

UN elemento clave de cualquier proceso de ingeniería es la medición. Empleamos medidas para entender mejor los atributos de los modelos que creamos. Pero, fundamentalmente, empleamos las medidas para valorar la calidad de los productos de ingeniería o de los sistemas que construimos.

Adiferencia de otras disciplinas, la ingeniería del software no está basada en leyes cuantitativas básicas de la Física. Las medidas absolutas, tales como el voltaje, la masa, la velocidad o la temperatura no son comunes en el mundo del software. En su lugar, intentamos obtener un conjunto de medidas indirectas que dan lugar a métricas que proporcionan una indicación de la calidad de algún tipo de representación del software. Como las medidas y métricas del software no son absolutas, están abiertas a debate. Fenton [FEN91] trata este aspecto cuando dice:

> La medición es el proceso por el que se asignan números o símbolos a los atributos de las entidades en el mundo real, de tal manera que las definan de acuerdo con unas reglas claramente definidas .... En las ciencias físicas, medicina y, más recientemente, en las ciencias sociales, somos ahora capaces de medir atributos que previamente pensábamos que no eran medibles ... Por supuesto, tales mediciones no están tan refinadas como las de las ciencias físicas..., pero existen [y se toman importantes decisiones basadas en ellas]. Sentimos que la obligación de intentar «medir lo no medible» para mejorar nuestra comprensión de entidades particulares es tan poderosa en la ingeniería del software como en cualquier disciplina.

Pero algunos miembros de la comunidad de software continúan argumentando que el software no es medible o que se deberían posponer los intentos de medición hasta que comprendamos mejor el software y los atributos que habría que emplear para describirlo. Esto es un error.

VISTAZO RÁPIDO

¿Qué es? Por su naturaleza, la ingeniería es una disciplina cuantitativa. Los ingenieros usan números para ayudarse en el diseño y cálculo del producto a construir. Hasta hace poco tiempo. los ingenieros del software disponían de pocas referencias cuantitativas en su trabajo —pero esto está cambiando—. Las métricas técnicas ayudan a los ingenieros del software a profundizar en el diseño y construcción de los productos que desarrollan.

¿Quién lo hace? Los ingenieros del software usan las métricas técnicas para ayudarse en el desarrollo de software de mayor calidad.

¿Por qué es importante? Siempre habrá elementos cualitativos para la creación del software. El problema estriba en que la valoración cualitativa puede no ser suficiente. Un ingeniero del software necesita criterios objetivos para guiarse en el diseño de datos, de la arquitectura, de las interfaces y de los componentes. El verificador necesita una referencia cuantitativa que le ayude en la selección de los casos de prueba y de sus objetivos. Las métricas técnicas facilitan una base para que el análisis, diseño, codificación y prueba puedan ser conducidas más objetivamente y valoradas más cuantitativamente.

¿Cuáles son los pasos? La primera etapa en el proceso de medida consiste en extraer las medidas y métricas del software que son apropiadas para la representación del software que está siendo considerado. A continuación, se requieren datos para extraer la formulación de métricas agregadas. Una vez calculadas, las métricas apropiadas son analizadas en base a directrices preestablecidas y datos históricos. El resultado del análisis es interpretado para profundizar en la calidad del software, y el resultado de la interpretación orienta las modificaciones a originarse en los resultados obtenidos en el análisis, diseño, codificación y prueba.

¿Cuál es el producto obtenido? Las métricas del software serán calculadas sobre datos agregados del análisis, de los modelos de diseño, del código fuente y de los casos de prueba.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Hay que establecer los objetivos y medidas antes de comenzar la acumulación de datos, definiendo sin ambigüedad cada métrica técnica. Define pocas métricas y utilízalas para profundizar en la calidad del resultado obtenido en la ingeniería del software.

Aunque las métricas técnicas para el software de computadora no son absolutas, nos proporcionan una manera sistemática de valorar la calidad basándose en un conjunto de «reglas claramente definidas». También le proporcionan al ingeniero del software una visión interna en el acto, en vez de a posteriori. Esto permite al ingeniero descubrir y corregir problemas potenciales antes de que se conviertan en defectos catastróficos.

En el Capítulo 4 estudiábamos las métricas del software tal y como se aplican a nivel del proceso y del proyecto. En este capítulo, nuestro punto de atención se desplaza a las medidas que se pueden emplear para valorar la calidad del producto según se va desarrollando. Estas medidas de atributos internos del producto le proporcionan al desarrollador de software una indicación en tiempo real de la eficacia del análisis, del diseño y de la estructura del código, la efectividad de los casos de prueba, y la calidad global del software a construir.

19.1 CALIDAD DEL SOFTWARE

Incluso los desarrolladores del software más hastiados estarán de acuerdo en que un software de alta calidad es una de las metas más importantes. Pero, ¿cómo definimos la calidad? En el Capítulo **8** propusimos diferentes maneras de interpretar la calidad del software e introdujimos una definición que hacía hincapié en la concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos, los estándares de desarrollo explícitamente documentados y las características implícitas que se esperan de todo software desarrollado profesionalmente.

Cita:
Todo programa hace algo correctamente, que puede no ser lo que necesitamos que haga.
Anónimo

No cabe duda de que la definición anterior podría modificase o ampliarse y discutirse eternamente. Para los propósitos de este libro, la definición sirve para hacer énfasis en tres puntos importantes:

1. Los requisitos del software son la base de las medidas de la calidad. La falta de concordancia con los requisitos es una falta de calidad[1].
2. Unos estándares específicos definen un conjunto de criterios de desarrollo que guían la manera en que se hace la ingeniería del software. Si no se siguen los criterios, habrá seguramente poca calidad.
3. Existe un conjunto de requisitos implícitos que a menudo no se nombran (por ejemplo, facilidad de mantenimiento). Si el software cumple con sus requisitos explícitos pero falla en los implícitos, la calidad del software no será fiable.

La calidad del software es una compleja mezcla de factores que variarán a través de diferentes aplicaciones y según los clientes que las pidan. En las siguientes secciones, se identifican los factores de la calidad del software y se describen las actividades humanas necesarias para conseguirlos.

19.1.1. Factores de calidad de McCall

Los factores que afectan a a la calidad del software se pueden categorizar en dos amplios grupos: (1) factores que se pueden medir directamente (por ejemplo, defectos por punto de función) y (2) factores que se pueden medir sólo indirectamente (por ejemplo, facilidad de uso o de mantenimiento). En todos los casos debe aparecer la medición. Debemos comparar el software (documentos, programas, datos) con una referencia y llegar a una conclusión sobre la calidad.



CLAVE

Es interesante anotar que los factores de calidad de McCall son tan válidos hoy como cuando fueron los primeros propuestos en los años 70. Además, es razonable indicar que los factores que afectan a la calidad del software no cambian.

McCall y sus colegas [MCC77] propusieron una útil clasificación de factores que afectan a la calidad del software. Estos factores de calidad del software, mostrados en la Figura 19.1, se concentran en tres aspectos importantes de un producto software: sus características operativas, su capacidad de cambios y su adaptabilidad a nuevos entornos.

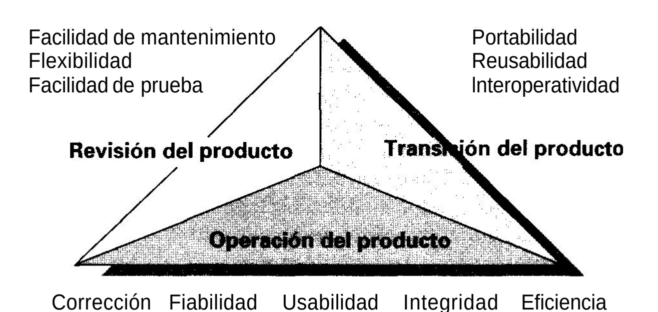


FIGURA 19.1. Factores de calidad de McCall

Refiriéndose a los factores anotados en la Figura 19.1, McCall proporciona las siguientes descripciones:

Corrección. Hasta dónde satisface un programa su especificación y logra los objetivos propuestos por el cliente.

Fiabilidad. Hasta dónde se puede esperar que un programa lleve a cabo su función con la exactitud requerida. Hay que hacer notar que se han propuesto otras definiciones de fiabilidad más completas (vea el Capítulo 8).

[1] Es importante resaltar que la calidad se extiende a los atributos técnicos de los modelos de análisis, de diseño y de codificación. Los modelos que presentan una alta calidad (en el sentido técnico) darán lugar a un software de una alta calidad desde el punto de vista del cliente.

Eficiencia. La cantidad de recursos informáticos y de código necesarios para que un programa realice su función.

Integridad. Hasta dónde se puede controlar el acceso al software o a **los** datos por personas no autorizadas.



Cita:
La calidad de un producto es una función de los muchos cambios del mundo por mejorar.
Tom DeMarco

Usabilidad (facilidad de manejo). El esfuerzo necesario para aprender a operar con el sistema, preparar **los** datos de entrada e interpretar las salidas (resultados) de un programa.

Facilidad de mantenimiento. El esfuerzo necesario para localizar y arreglar un error en un programa. (Esta es una definición muy limitada).

Flexibilidad. El esfuerzo necesario para modificar un programa que ya está en funcionamiento.

Facilidad de prueba. El esfuerzo necesario para probar un programa y asegurarse de que realiza correctamente su función.

Portabilidad. El esfuerzo necesario para transferir el programa de un entorno hardware/software a otro entomo diferente.

Reusabilidad (capacidad de reutilización). Hasta dónde se puede volver a emplear un programa (o partes de un programa) en otras aplicaciones, en relación al empaquetamiento y alcance de las funciones que realiza el programa.

Interoperatividad. **El** esfuerzo necesario para acoplar un sistema con otro.

Es difícil, y en algunos casos imposible, desarrollar medidas directas de los factores de calidad anteriores. Por tanto, se definen y emplean un conjunto de métricas para desarrollar expresiones para todos los factores, de acuerdo con la siguiente relación:

$$F_q = c_1 \times m_1 + c_2 \times m_2 + ... + c_n \times m_n$$

donde F_q es un factor de calidad del software, c, son coeficientes de regresión y m, son las métricas que afectan al factor de calidad. Desgraciadamente, muchas de las métricas definidas por McCall pueden medirse solamente de manera subjetiva. Las métricas pueden ir en forma de lista de comprobación que **se** emplea para «puntuar» atributos específicos del software [CAV78]. El esquema de puntuación propuesto por McCall es una escala del 0 (bajo) al 10 (alto). Se emplean las siguientes métricas en el esquema de puntuación:

Referencia cruzada
Las métricas **indicadas** pueden ser valoradas en las revisiones técnicas formales ***referenciadas*** en **el** Capítulo 8.

Facilidad de auditoría. La facilidad con la que se puede comprobar el cumplimeinto de **los** estándares.

Exactitud. La exactitud de **los** cálculos y del control.

Estandarización de comunicaciones. El grado de empleo de estándares de interfaces, protocolos y anchos de banda.

Complección. El grado con que se ha logrado la implementación total de una función.

Concisión. Lo compacto que es el programa en términos de líneas de código.

Consistencia. El empleo de un diseño uniforme y de técnicas de documentación **a** lo largo del proyecto de desarrollo del software.

Estandarización de datos. El empleo de estructuras y tipos de datos estándares a lo largo del programa.

Tolerancia al error. El daño causado cuando un programa encuentra un error.

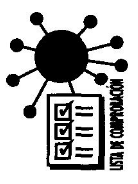
Eficiencia de ejecución. **El** rendimiento del funcionamiento de un programa.

Capacidad de expansión. El grado con que se pueden ampliar el diseño arquitectónico, de datos o procedimental.

Generalidad. La amplitud de aplicación potencial de **los** componentes del programa.

Independencia del hardware. El grado con que se desacopla el software del hardware donde opera.

Instrumentación. El grado con que el programa vigila su propio funcionamiento e identifica los errores que ocurren.



Factores de Calidad

Modularidud. La independencia funcional (Capítulo **13)** de componentes de programa.

Operatividud. La facilidad de operación de un programa.

Seguridad. La disponibilidad de mecanismos que controlan o protegen los programas y los datos.

Autodocumentación. El grado en que el código fuente proporciona documentación significativa.

Simplicidad. **El** grado de facilidad con que se puede entender un programa.

Independencia del sistema software. El grado de independencia de programa respecto a las características del lenguaje de programación no estándar, características del sistema operativo y otras restricciones del entorno.

Trazabilidad. La capacidad de seguir una representación del diseño o un componente real del programa hasta **los** requisitos.

Formación. **El** grado en que ayuda el software a manejar el sistema a los nuevos usuarios.

La relación entre los factores de calidad del software y las métricas de la lista anterior se muestra en la Figura 19.2. Debería decirse que el peso que se asigna a cada métrica depende de los productos y negocios locales.

19.1.2. FURPS

Los factores de calidad descritos por McCall y **sus** colegas [MCC77] representan sólo una de las muchas listas de comprobación sugeridas para la calidad del software. Hewlett-Packard [GRA87] ha desarrollado un conjunto de factores de calidad del software al que se le ha

dado el acrónimo de ***FURPS:*** funcionalidad, *facilidad* de *uso,* fiabilidad, rendimiento y capacidad de soporte. Los factores de calidad FURPS provienen de trabajos anteriores, definiendo los siguientes atributos para cada uno de los cinco factores principales:

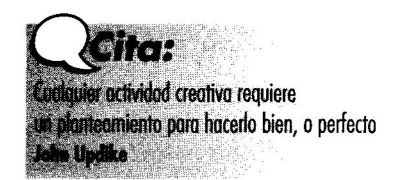
- La *funcionalidad* se valora evaluando el conjunto de características y capacidades del programa, la generalidad de las funciones entregadas y la seguridad del sistema global.
- La facilidad de uso se valora considerando factores humanos (capítulo 15), la estética, la consistencia y la documentación general.
- La *fiabilidad* se evalúa midiendo la frecuencia y gravedad de los fallos, la exactitud de las salidas (resultados), el tiempo de medio de fallos (TMDF), la capacidad de recuperación de un fallo y la capacidad de predicción del programa.
- El rendimiento se mide por la velocidad de procesamiento, el tiempo de respuesta, consumo de recursos, rendimiento efectivo total y eficacia.
- La capacidad de soporte combina la capacidad de ampliar **el** programa (extensibilidad), adaptabilidad y servicios (estos tres atributos representan un término más común —mantenimiento—), así como capacidad de hacer pruebas, compatibilidad, capacidad de configuración (la capacidad de organizar y controlar elementos de la configuración del software [Capítulo 9]), la facilidad de instalación de un sistema y la facilidad con que se pueden localizar los problemas.

Los factores de calidad FURPS **y** atributos descritos anteriormente pueden usarse para establecer métricas de la calidad para todas las actividades del proceso del software.

19.1.3. Factores de calidad ISO 9126

El estándar ISO 9126 ha sido desarrollado en un intento de identificar los atributos clave de calidad para el software. El estándar identifica seis atributos clave de calidad:

> *Funcionalidad.* **El** grado en que el software satisface las necesidades indicadas por los siguientes subatributos: idoneidad, corrección, interoperatividad, conformidad y seguridad.
>
> *Confiabilidad.* Cantidad de tiempo que el software está disponible para su uso. Está referido por los siguientes subatributos: madurez, tolerancia a fallos y facilidad de recuperación.
>
> *Usabilidad.* Grado en que el software es fácil de usar. Viene reflejado por los siguientes subatributos: facilidad de comprensión, facilidad de aprendizaje y operatividad.
>
> *Eficiencia.* Grado en que el software hace Óptimo el uso de los recursos del sistema. Está indicado por los siguientes subatributos: tiempo de uso y recursos utilizados.
>
> *Facilidad de mantenimiento.* La facilidad con que una modificación puede ser realizada. Está indicada por los siguientes subatributos: facilidad de análisis, facilidad de cambio, estabilidad **y** facilidad de prueba.
>
> *Portabilidad.* La facilidad con que el software puede ser llevado de un entorno a otro. Está referido por los siguientes subatributos: facilidad de instalación, facilidad de ajuste, facilidad de adaptación al cambio.



Del mismo modo que los factores de calidad estudiados en las secciones 19.1.1. y 19.1.2., los factores ISO 9126 no necesariamente son utilizados para medidas directas. En cualquier caso, facilitan una valiosa base para medidas indirectas **y** una excelente lista para determinar la calidad de un sistema.

Métrica de la calidad del software / Factor de calidad	Corrección	Fiabilidad	Eficiencia	Integridad	Mantenimiento	Flexibilidad	Capacidad de pruebas	Portabilidad	Reusabilidad	Interoperatividad	Usabilidad
Facilidad de auditoría				X			X				
Exactitud		X									
Estandarización de comunicaciones										X	
Compleción	X										
Complejidad		X				X	X				
Concisión			X		X	X					
Consistencia	X	X			X	X					
Estandarización de datos										X	
Tolerancia a errores		X									
Eficiencia de ejecución			X								
Capacidad de expansión						X					
Generalidad						X		X	X	X	
Independiencia del hardware								X	X		
Instrumentación				X	X		X				
Modularidad		X			X	X	X	X	X	X	
Operatividad			X								X
Seguridad				X							
Autodocumentación					X	X	X	X	X		
Simplicidad		X			X	X	X				
Independencia del sistema								X	X		
Trazabilidad	X										
Facilidad de formación											X

FIGURA 19.2. Factores y métricas de calidad.

19.1.4. La transición a una visión cuantitativa

En las secciones precedentes se estudiaron un conjunto de factores cualitativos para la «medición» de la calidad del software. Intentamos desarrollar medidas exactas de la calidad del software frustradas a veces por la naturaleza subjetiva de la actividad. Cavano y McCall [CAV78] estudian esta situación:

> La determinación de la calidad es un factor clave en los acontecimientos diarios: concursos de cata de vinos, acontecimientos deportivos (por ejemplo, la gimnasia), concursos de talento, etc. En estas situaciones, la calidad se juzga de la manera más fundamental y directa: comparación de objetos unos al

lado de los **otros** bajo condiciones idénticas y con conceptos predeterminados. El vino puede ser juzgado de acuerdo con **su** claridad, color, bouquet, sabor, etc. **Sin** embargo, este tipo de juicio es muy subjetivo; para que tenga algo de valor, debe hacerse por un experto.

La subjetividad y la especialización también influyen en la determinación de la calidad del software. Para resolver este problema, se necesita una definición de calidad del software más exacta, así como una manera de obtener medidas cuantitativas de la calidad del software para hacer un análisis objetivo.... Como no existe el conocimiento absoluto, no deberíamos esperar poder medir la calidad del software exactamente, ya que cada medición es parcialmente imperfecta. Jacob Bronkowski describió esta paradoja del conocimiento de la siguiente manera: «Año **tras** año ingeniamos instrumentos más exactos con los que observar la naturaleza con más exactitud. **Y** cuando miramos las observaciones estamos desconcertados de ver que todavía son confusas, y tenemos la sensación de que son tan inciertas como siempre.»

En las siguientes secciones examinamos un conjunto de métricas del software que pueden aplicarse a la valoración cuantitativa de la calidad del software. En todos los casos, las métricas representan medidas indirectas; es decir, realmente nunca medimos la calidad sino alguna manifestación de la calidad. El factor que lo complica es la relación exacta entre la variable que se mide y la calidad del software.

19.2 UNA ESTRUCTURA PARA LAS MÉTRICAS TÉCNICAS DEL SOFTWARE

Como dijimos en la introducción de este capítulo, la medición asigna números o símbolos a atributos de entidades en el mundo real. Para conseguirlo se necesita un modelo de medición que comprenda un conjunto consistente de reglas. Aunque la teoría de la medición (por ejemplo, [KYB84]) y su aplicación al software (por ejemplo, [DEM81], [BRI96]) son temas que están más allá del alcance de este libro, merece la pena establecer una estructura fundamental y un conjunto de principios básicos para la medición de métricas técnicas para el software.

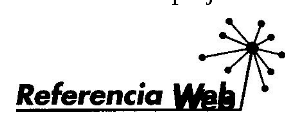


Lo mismo que las medidas de temperatura comienzan con un índice de referencia y evolucionan por sofisticadas escalas, herramientas y técnicas, de la misma forma están madurando las medidas del software.

Shari Pfleeger

19.2.1. El reto de las métricas técnicas

Durante las pasadas tres décadas, muchos investigadores han intentado desarrollar una sola métrica que proporcione una medida completa de la complejidad del software. Fenton [FEN94] caracteriza esta investigación como una búsqueda del «imposible santo grial». Aunque se han propuesto docenas de medidas de complejidad [ZUS90], cada una tiene un punto de vista diferente de lo que es la complejidad y qué atributos de un sistema llevan a la complejidad.



Voluminosa información sobre métricas técnicas han sido recopiladas por Horst Zuse: **irb.cs.tu-berlin.de/~zuse/**

Pero existe la necesidad de medir y controlar la complejidad del software. Y si es difícil de obtener un solo valor de esta «métrica de calidad», si debería ser posible desarrollar medidas de diferentes atributos internos del programa (por ejemplo, modularidad efectiva, independencia funcional y otros atributos tratados desde el Capítulo 13 al Capítulo 16). Estas medidas **y** las métricas obtenidas de ellas pueden usarse como indicadores independientes de la calidad de los modelos de análisis y diseño. Pero también surgen problemas aquí. Fenton [FEN94] lo advierte cuando dice:

> El peligro de intentar encontrar medidas que caractericen tantos atributos diferentes es que, inevitablemente, las medidas tienen que satisfacer objetivos incompatibles. Esto es contrario a la teoría de representación de la medición.

Aunque la declaración de Fenton es correcta, mucha gente argumenta que la medición técnica llevada a cabo durante las primeras fases del proceso de software les proporciona a los desarrolladores de software un consistente **y** objetivo mecanismo para valorar la calidad.

Conviene preguntarse, no obstante, qué validez tienen las métricas técnicas. **Es** decir, ¿cómo están de próximas las métricas técnicas **y** la fiabilidad y la calidad a largo plazo de un sistema basado en computadora? Fenton [FEN91] trata esta cuestión de la siguiente manera:

> A pesar de las intuitivas conexiones entre la estructura interna de los productos software (métricas técnicas) y **su** producto externo, y los atributos del proceso, ha habido, de hecho, muy pocos intentos científicos para establecer relaciones específicas. Hay varias razones para ello; la que se menciona más a menudo es la imposibilidad de llevar a cabo experimentos.

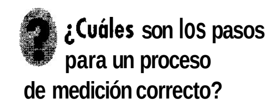
Todos los desafíos mencionados anteriormente son motivo de precaución, pero no es motivo de desprecio de las métricas técnicas[2]. La medición es esencial si se desea conseguir calidad.

[2] Se ha producido una gran cantidad de literatura sobre las métricas del software (por ejemplo, vea [FEN94], [ROC94], [ZUS97] para obtener unas extensas bibliografías) y es común la crítica de métricas específicas (incluyendo algunas de las presentadas en este capítulo). Sin embargo, muchas de las críticas se concentran en aspectos esotéricos y pierden el objetivo primario de la medición en el mundo real: ayudar al ingeniero a establecer una manera sistemática y objetiva de conseguir una visión interna de su trabajo y mejorar la calidad del producto como resultado.

19.2.2. Principios de medición

Antes de introducir una serie de métricas técnicas que **(1)** ayuden a la evaluación de los modelos de análisis y diseño, (2) proporcionen una indicación de la complejidad de los diseños procedimentales y del código fuente, y **(3)** ayuden en el diseño de pruebas más efectivas, es importante entender los principios básicos de la medición. Roche [ROC94] sugiere un proceso de medición que se puede caracterizar por cinco actividades:

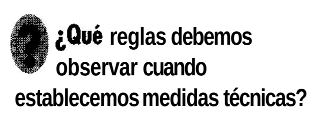
- *formulación:* la obtención de medidas y métricas del software apropiadas para la representación del software en cuestión.
- *colección:* el mecanismo empleado para acumular datos necesarios para obtener las métricas formuladas.
- *análisis:* el cálculo de las métricas y la aplicación de herramientas matemáticas.



- *interpretación:* la evaluación de los resultados de las métricas en un esfuerzo por conseguir una visión interna de la calidad de la representación.
- *realimentación (feedback):* recomendaciones obtenidas de la interpretación de métricas técnicas transmitidas al equipo que construye el software.

Los principios que se pueden asociar con la formulación de las métricas técnicas son los siguientes [ROC94]:

- Los objetivos de la medición deberían establecerse antes de empezar la recogida de datos.
- Todas las técnicas sobre métricas deberían definirse sin ambigüedades.



- Las métricas deberían obtenerse basándose en una teoría válida para el dominio de aplicación (por ejemplo, las métricas para el diseño han de dibujarse sobre conceptos y principios básicos de diseño y deberían intentar proporcionar una indicación de la presencia de un atributo que se considera beneficioso).
- Hay que hacer las métricas a medida para acomodar mejor los productos y procesos específicos [BAS84].

Aunque la formulación es un punto de arranque crítico, la recogida y análisis son las actividades que dirigen el proceso de medición. Roche [ROC94] sugiere los siguientes principios para estas actividades:

- Siempre que sea posible, la recogida de datos y el análisis debe automatizarse.



Por encima de todo, intento obtener medidos técnicas simples. No te obsesiones por lo métrica «perfecta» porque no existe.

- Se deberían aplicar técnicas estadísticas válidas para establecer las relaciones entre los atributos internos del producto y las características externas de la calidad (por ejemplo, ¿está correlacionado el nivel de complejidad arquitectónico con el número de defectos descubiertos en la producción?).
- Se deberían establecer una directrices de interpretación y recomendaciones para todas las métricas.

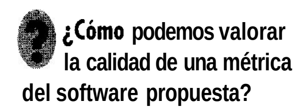
Además de los principios apuntados anteriormente, el éxito de una actividad de métrica está ligada al soporte de gestión. Se deben considerar los fondos, la formación y la promoción si se quiere establecer y mantener un programa de medición técnica.

19.2.3. Características fundamentales de las métricas del software

Se han propuesto cientos de métricas para el software, pero no todas proporcionan un soporte práctico para el desarrollador de software. Algunas demandan mediciones que son demasiado complejas, otras son tan esotéricas que pocos profesionales tienen la esperanza de entenderlas y otras violan las nociones básicas intuitivas de lo que realmente es el software de alta calidad.

Ejiogu [EJI91] define un conjunto de atributos que deberían acompañar a las métricas efectivas del software. La métrica obtenida y las medidas que conducen a ello deberían ser:

- *simples y fáciles de calcular.* Debería ser relativamente fácil aprender a obtener la métrica y su cálculo no debería demandar un esfuerzo o cantidad de tiempo inusuales.



- *empírica e intuitivamente persuasivas.* La métrica debería satisfacer las nociones intuitivas del ingeniero sobre el atributo del producto en cuestión (por ejemplo, una métrica que mide la cohesión de un módulo debería aumentar su valor a medida que crece el nivel de cohesión).
- *consistentes* y *objetivas.* La métrica debería siempre producir resultados sin ambigüedad. Un tercer equipo debería ser capaz de obtener el mismo valor de métrica usando la misma información del software.
- *consistentes en el empleo de unidades y tamaños.* El cálculo matemático de la métrica debería emplear medidas que no conduzcan a extrañas combinaciones de unidades. Por ejemplo, multiplicando el número de personas de un equipo por las variables del lenguaje de programación en el programa resulta una sospechosa mezcla de unidades que no son intuitivamente persuasivas.
- *independientes del lenguaje de programación.* Las métricas deberían basarse en el modelo de análisis, modelo de diseño o en la propia estructura del

programa. No deberían depender de los caprichos de la sintaxis o semántica del lenguaje de programación.

- *un eficaz mecanismo para la realimentación de calidad.* La métrica debería proporcionar, al desarrollador de software, información que le lleve a un producto final de mayor calidad.



*La experiencia indica que una métrica técnica se usa únicamente si es **intuitiva** y **fácil** de calcular. Si se requiere **docenas** de **«contadores»** y se han de utilizar **complejos cálculos**, la métrica **no será** ampliamente utilizado.*

Aunque la mayoría de las métricas de software satisfacen las características anteriores, algunas de las métricas comúnmente empleadas dejan de cumplir una o dos. Un ejemplo es el punto de función (tratado en el Capítulo **4** y en este capítulo). Se puede argumentar[3] que el atributo consistente y objetivo falla porque un equipo ajeno independiente puede no ser capaz de obtener el mismo valor del punto de función que otro equipo que use la misma información del software. ¿Deberíamos entonces rechazar la medida PF? La respuesta es «¡por supuesto que no!». El PF proporciona una útil visión interna y por tanto proporciona un valor claro, incluso si no satisface un atributo perfectamente.

19.3 MÉTRICAS DEL MODELO DE ANÁLISIS

El trabajo técnico en la ingeniería del software empieza con la creación del modelo de análisis. En esta fase se obtienen los requisitos y se establece el fundamento para el diseño. Por tanto, son deseables las métricas técnicas que proporcionan una visión interna a la calidad del modelo de análisis.

Referencia cruzada

Los modelos de datos, funciones **y** comportamientos son tratados en los **Capítulos** 11 **y 12.**

Aunque han aparecido en la literatura relativamente pocas métricas de análisis y especificación, es posible adaptar métricas obtenidas para la aplicación de un proyecto (Capítulo **4**) para emplearlas en este contexto. Estas métricas examinan el modelo de análisis con la intención de predecir el «tamaño» del sistema resultante. Es probable que el tamaño y la complejidad del diseño estén directamente relacionadas.

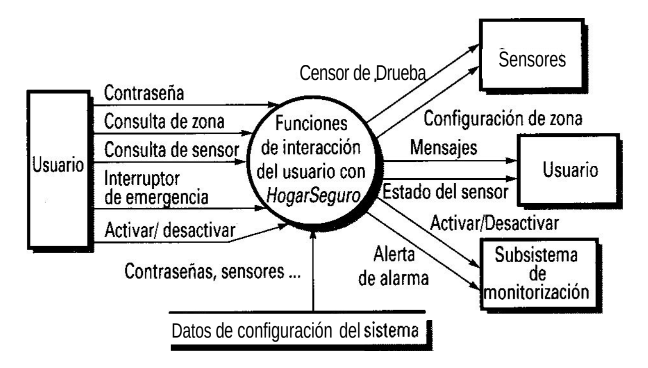


FIGURA 19.3. Parte del modelo de análisis del software de *Hogar Seguro.*

19.3.1. Métricas basadas en la función

La métrica de punto de función (PF) (Capítulo **4**) se puede usar como medio para predecir el tamaño de un sistema que se va a obtener de un modelo de análisis. Para ilustrar el empleo de la métrica de PF en este contexto, consideramos una sencilla representación del modelo de análisis mostrada en la Figura 19.3. En la figura se representa un diagrama de flujo de datos (Capítulo 12) de una función del sistema *HogarSeguro*[4]. La función gestiona la interacción con el usuario, aceptando una contraseña de usuario para activar/desactivar el sistema y permitiendo consultas sobre el estado de las zonas de seguridad y varios sensores de seguridad. La función muestra una serie de mensajes de petición y envía señales apropiadas de control a varios componentes del sistema de seguridad.



*Para ser útil para el **trabajo** técnica, las medidas técnicas que apoyan la **toma** de decisión **(ej: errores encontrados** durante la prueba de unidad) deben ser agrupadas y normalizadas **utilizando** la métrica PF.*

El diagrama de flujo de datos se evalúa para determinar las medidas clave necesarias para el cálculo de la métrica de punto de función (Capítulo **4**):

- número de entradas del usuario
- número de salidas del usuario
- número de consultas de usuario
- número de archivos
- número de interfaces externas

Tres entradas del usuario: **contraseña, interruptor de emergencia** y **activar/desactivar** aparecen en la figura junto con dos consultas: **consulta de zona** y **consulta de sensor.** Se muestra un archivo **(archivo de**

[3] Fíjese que se puede hacer un contra argumento igualmente vigoroso. Tal es la naturaleza de las métricas del software

[4] *HogarSeguro* es un sistema de seguridad para el hogar que se ha usado como aplicación de ejemplo en capítulos anteriores

configuración del sistema). También están presentes dos salidas de usuarios **(mensajes** y **estados del sensor)** y cuatro interfaces externas **(sensor de prueba, configuración de zona, activar/desactivar** y **alerta de alarma).** Estos datos, junto con la complejidad apropiada, se muestran en la Figura 19.4.

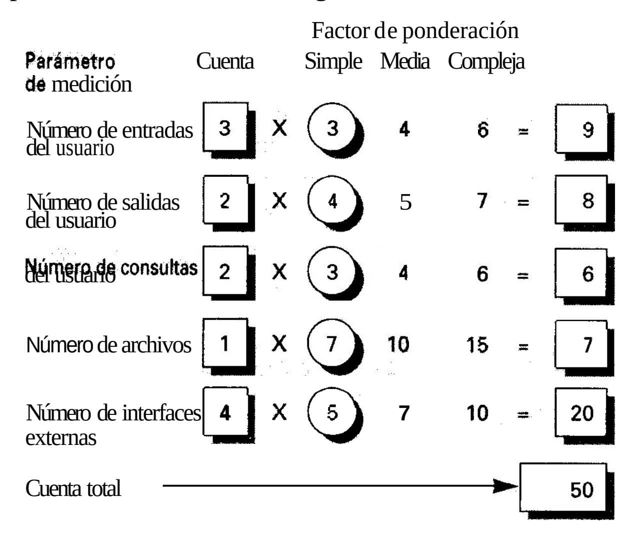


FIGURA 19.4. Cálculo de puntos de función para una función de Hogar *Seguro.*

La *cuenta total* mostrada en la Figura 19.4 debe ajustarse usando la ecuación (4.1):

$$PF = \text{cuenta-total} \times (0{,}65 + 0{,}01 \times \sum[Fi])$$

Donde cuenta-total es la suma de todas las entradas PF obtenidas en la Figura 19.3 y F_i (i=1 a 14) son «los valores de ajuste de complejidad». Para el propósito de este ejemplo, asumimos que $\sum[Fi]$ es 46 (un producto moderadamente complejo). Por tanto:

$$PF = 50 \times [0{,}65 + (0{,}01 \times 46)] = 56$$



Una introducción útil al PF ha sido elaborada por Capers Jones y puede ser localizada en: **www.spr.com/ library/ Ofuncmet.htm**

Basándose en el valor previsto de PF obtenido del modelo de análisis, el equipo del proyecto puede estimar el tamaño global de implementación de las funciones de interacción de *HogarSeguro.* Asuma que los datos de los que se disponen indican que un PF supone 60 líneas de código (se va a usar un lenguaje orientado a objetos) y que en un esfuerzo de un mes-persona se producen 12 PF. Estos datos históricos proporcionan al gestor del proyecto una importante información de planificación basada en el modelo de análisis en lugar de en estimaciones preliminares.

Considerar que de los proyectos anteriores se han encontrado una media de **3** errores por punto de función durante las revisiones de análisis y diseño, y 4 errores por punto de función durante las pruebas unitaria y de integración. Estos datos pueden ayudar a valorar a los ingenieros del software la completitud de sus revisiones y las actividades de prueba.

19.3.2. La Métrica *bang*

Al igual que la métrica de punto de función, la *métrica bang* puede emplearse para desarrollar una indicación del tamaño del software a implementar como consecuencia del modelo de análisis.

Desarrollada por DeMarco [DEM82], la métrica *bang* es «una indicación independiente de la implementación del tamaño del sistema». Para calcular la métrica *bang,* el desarrollador de software debe evaluar primero un conjunto de primitivas (elementos del modelo de análisis que no se subdividen más en el nivel de análisis). Las primitivas [DEM82] se determinan evaluando el modelo de análisis y desarrollando cuentas para los siguientes elementos?

Primitivas funcionales (PFu). Transformaciones (burbujas) que aparecen en el nivel inferior de un diagrama de flujo de datos (Capítulo 12).

Elementos de datos (ED). Los atributos de un objeto **de** datos, los elementos de datos son datos no compuestos y aparecen en el diccionario de datos.

Objetos (OB). Objetos de datos tal y como se describen **en** el Capítulo 11.

Relaciones (RE). Las conexiones entre objetos de datos tal y como se describen en el Capítulo 12.

Estados (ES). El número de estados observables **por** el usuario en el diagrama de transición de estados (Capítulo **12).**

Transiciones (TR). El número de transiciones de estado en el diagrama de transición de estados (Capítulo 12).



Antes que reflexionemos sobre una «nueva métrica» debemos aplicar ... podemos preguntarnos nosotros mismos sobre los puntos básicos, «Qué pretendemos con las métricas.
Michael Mah y Larry Putnam

Además de las seis primitivas apuntadas arriba, se determinan las cuentas adicionales para:

Primitivas modificadas de función manual (PMFu). Funciones que caen fuera del límite del sistema y que deben modificarse para acomodarse al nuevo sistema.

5 El acrónimo apuntado entre paréntesis a continuación de la primitiva se emplea para denotar **la** *cuenta* de la primitiva particular; por ejemplo, PFu indica el número de primitivas funcionales presente en un modelo de análisis.

Elementos de datos de entrada (EDE). Aquellos elementos de datos que se introducen en el sistema.

Elementos de datos de salida (EDS). Aquellos elementos de datos que se sacan del sistema.

Elementos de datos retenidos (EDR). Aquellos elementos de datos que son retenidos (almacenados) por el sistema.

Muestras *(tokens)* de datos (TC_i). Las muestras de datos (elementos de datos que no se subdividen dentro de una primitiva funcional) que existen en el límite de la i-ésima primitiva funcional (evaluada para cada primitiva).

Conexiones de relación (RE_i). Las relaciones que conectan el i-ésimo objeto en el modelo de datos con otros objetos.

DeMarco [DEM82] sugiere que la mayoría del software se puede asignar a uno de los dos dominios siguientes, ***dominio de función o dominio de datos***, dependiendo de la relación RE/PFu. Las aplicaciones de dominio de función (encontradas comúnmente en aplicaciones de ingeniería y científicas) hacen hincapié en la transformación de datos y no poseen generalmente estructuras de datos complejas. Las aplicaciones de dominio de datos (encontradas comúnmente en aplicaciones de sistemas de información) tienden a tener modelos de datos complejos.

RE/PFu < 0,7 implica una aplicación de dominio de función

0,8 < RE/PFu < 1,4 indica una aplicación híbrida

RE/PFu > 1,5 implica una aplicación de dominio de datos

Como diferentes modelos de análisis harán una partición del modelo con mayor o menor grado de refinamiento. DeMarco sugiere que se emplee una cuenta media de muestra ***(token)*** por primitiva

$$TC_{avg} = \Sigma TC_i / PFu$$

para controlar la uniformidad de la partición a través de muchos diferentes modelos dentro del dominio de una aplicación.

Para calcular la métrica *bang* para aplicaciones de dominio de función, se emplea el siguiente algoritmo:

```
Asignar a bang un valor inicial = ∅;
Mientras queden primitivas funcionales por evaluar
    Calcular cuenta-token alrededor del límite de la
    primitiva i;
    Calcular el incremento PFu corregido (IPFuC)
    Asignar la primitiva a una clase
    Evaluar la clase y anotar el peso valorado
    Multiplicar IPFuC por el peso valorado
    bang = bang + ponderación IPFuC;
FinMientras
```

La cuenta-token se calcula determinando cuántos símbolos léxicos ***(tokens)*** diferentes son «visibles» [DEM82] dentro de la primitiva. Es posible que el número de símbolos léxicos ***(tokens)*** y el número de elementos de datos sea diferente, si los elementos de datos pueden moverse desde la entrada a la salida sin ninguna transformación interna. La IPFuC corregida se determina de una tabla publicada por DeMarco. A continuación, se presenta una versión muy abreviada:

Tci	IPFuC
2	1,0
5	5,8
10	16,6
15	29,3
20	43,2

La ponderación valorada apuntada en el algoritmo anterior se calcula de dieciséis clases diferentes de primitivas funcionales definidas por DeMarco. Se asigna una ponderación que va de 0,6 (encaminamiento simple de datos) a 2,5 (funciones de gestión de datos) dependiendo de la clase de la primitiva.

Para aplicaciones de dominio de datos, se calcula la métrica ***bang*** mediante el siguiente algoritmo:

```
Asignar a bang el valor inicial = ∅;
Mientras queden objetos por evaluar en el modelo de
datos
    Calcular la cuenta de relaciones' del objeto i
    Calcular el incremento de OB corregido (IOBC);
    bang = bang + IOBC;
FinMientras
```

El IOBC corregido se determina también de una tabla publicada por DeMarco. A continuación se muestra una versión abreviada:

REi	IOBC
1	1,0
3	4,0
6	9,0

Una vez que se ha calculado la métrica ***bang,*** se puede emplear el historial anterior para asociarla con el esfuerzo y el tamaño. DeMarco sugiere que las organizaciones se construyan sus propias versiones de tablas IPFuC e IOBC para calibrar la información de proyectos completos de software.

19.3.3. Métricas de la calidad de la especificación

Davis y sus colegas [DAV93] proponen una lista de características que pueden emplearse para valorar la calidad del modelo de análisis y la correspondiente especificación de requisitos: especificidad (ausencia de ambigüedad), compleción, corrección, comprensión, capacidad de verificación, consistencia interna y externa, capacidad de logro, concisión, trazabilidad, capacidad de modificación, exactitud y capacidad de reutilización. Además, los autores apuntan que las especificaciones de alta calidad deben estar almacenadas electrónicamente, ser ejecutables o al menos interpretables, anotadas por importancia y estabilidad relativas, con su versión correspondiente, organizadas, con referencias cruzadas y especificadas al nivel correcto de detalle.

Aunque muchas de las características anteriores parecen ser de naturaleza cualitativa, Davis [DAV93] sugie-

re que todas puedan representarse usando una o más métricas[6]. Por ejemplo, asumimos que hay n_r requisitos en una especificación, tal como

$$n_r = n_f + n_{nf}$$

donde n_f es el número de requisitos funcionales y n_{nf} es el número de requisitos no funcionales (por ejemplo, rendimiento).



Para medir las característicasde la especificación, es necesario conseguir profundizar **cuantitativamente** en la especificidad y en la **completitud**.

Para determinar **la** *especificidad* (ausencia de ambigüedad) de los requisitos. Davis sugiere una métrica basada en la consistencia de la interpretación de los revisores para cada requisito:

$$Q_1 = n_{ui} / n_r$$

donde n_{ui} es el número de requisitos para los que todos los revisores tuvieron interpretaciones idénticas. Cuanto más cerca de 1 esté el valor de Q, menor será la ambigüedad de la especificación.

La *compleción* de los requisitos funcionales pueden determinarse calculando la relación

$$Q_2 = u_n / (n_i \times n_s)$$

donde u, es el número de requisitos únicos de función, n_i es el número de entradas (estímulos) definidos o implicados por la especificación y n_s es el número de estados especificados. La relación Q, mide el porcentaje de funciones necesarias que se han especificado para un sistema. Sin embargo, no trata los requisitos no funcionales. Para incorporarlos a una métrica global completa, debemos considerar el grado de validación de los requisitos.

$$Q_3 = n_c / (n_c + n_{nv})$$

donde n_c es el número de requisitos que se han validado como correctos y n_{nv} el número de requisitos que no se han validado todavía.

19.4 MÉTRICAS DEL MODELO DE DISEÑO

Es inconcebible que el diseño de un nuevo avión, un nuevo chip de computadora o un nuevo edificio de oficinas se realizara sin definir las medidas del diseño, sin determinar las métricas para varios aspectos de la calidad del diseño **y** usarlas para guiar la evolución del diseño. Y sin embargo, el diseño de sistemas complejos basados en computadora a menudo consigue proseguir sin virtualmente ninguna medición. La ironía es que las métricas de diseño para el software están disponibles, pero la gran mayoría de los desarrolladores de software continúan sin saber que existen.

Cita:
La medida que es apreciable, y que no es apreciable, se hace apreciable.
Galileo

Las métricas de diseño para el software, como otras métricas del software, no son perfectas. Continúa el debate sobre la eficacia y cómo deberían aplicarse. Muchos expertos argumentan que se necesita más experimentación hasta que se puedan emplear las métricas de diseño. Y sin embargo, el diseño sin medición es una alternativa inaceptable.

En las siguientes secciones examinamos algunas de las métricas de diseño más comunes para el software de computadora. Aunque ninguna es perfecta, todas pueden proporcionar al diseñador una mejor visión interna y ayudar a que el diseño evolucione a un nivel superior de calidad.

19.4.1. Métricas del diseño arquitectónico

Las métricas de diseño de alto nivel se concentran en las características de la arquitectura del programa (Capítulo 14) con especial énfasis en la estructura arquitectónica y en la eficiencia de los módulos. Estas métricas son de caja negra en el sentido que no requieren ningún conocimiento del trabajo interno de un módulo en particular del sistema.

Card y Glass [CAR90] definen tres medidas de la complejidad del diseño del software: complejidad estructural, complejidad de datos y complejidad del sistema.

La *complejidad estructural*, $S(i)$, de un módulo *i* se define de la siguiente manera:

$$S(i) = f^2_{out}(i) \quad (19\text{-}1)$$

donde $f_{out}(i)$ es la expansión[7] del módulo i.

[6] Un estudio completo de las métricas de calidad de especificación está más allá del alcance de este capítulo. Vea [DAV93] para más detalles.

[7] Como se dijo en el estudio introducido en el Capítulo 13, la expansión (f_{out}) indica el número de módulos inmediatamente subordinados al módulo i, es decir, el número de módulos que son invocados directamente por el módulo i.



Las métricas pueden profundizar en la estructura, en los datos, y en la complejidad del sistema asociado con el diseño arquitectónico.

La *complejidad de datos, D(i),* proporciona una indicación de la complejidad en la interfaz interna de un módulo *i* y se define como:

$$D(i) = v(i) / [f_{out}(i) + 1] \quad (19.2)$$

donde *v(i)* es el número de variables de entrada y salida que entran y salen del módulo *i*.

Finalmente, la *complejidad del sistema, C(i)*, se define como la suma de las complejidades estructural y de datos, y se define como:

$$C(i) = S(i) + D(i) \quad (19.3)$$

A medida que crecen los valores de complejidad, la complejidad arquitéctónica o global del sistema también aumenta. Esto lleva a una mayor probabilidad de que aumente el esfuerzo necesario para la integración y las pruebas.



¿Hay un camino para valorar la Complejidad de ciertos modelos arquitectónicos?

Una métrica de diseño arquitectónico de alto nivel, propuesta por Henry y Kafura [HEN81], también emplea la expansión y la concentración. Los autores definen una métrica de complejidad de la forma:

$$MHK = longitud(i) \times [f_{in}(i) + f_{out}(i)]^2 \quad (19.4)$$

donde la *longitud(i)* es el número de sentencias en lenguaje de programación en el módulo i y $f_{in}(i)$ es la concentración del módulo *i*. Henry y Kafura amplían la definición de concentración y expansión presentadas en este libro para incluir no sólo el número de conexiones de control del módulo (llamadas al módulo), sino también el número de estructuras de datos del que el módulo i recoge (concentración) o actualiza (expansión) datos. Para calcular el MHK durante el diseño puede emplearse el diseño procedimental para estimar el número de sentencias del lenguaje de programación del módulo *i*. Como en las métricas de Card y Glass mencionadas anteriormente, un aumento en la métrica de Henry-Kafura conduce a una mayor probabilidad de que también aumente el esfuerzo de integración y pruebas del módulo.

Fenton [FEN91] sugiere varias métricas de morfología simples (por ejemplo, forma) que permiten comparar diferentes arquitecturas de programa mediante un conjunto de dimensiones directas. En la Figura 19.5, se pueden definir las siguientes métricas:

$$tamaño = n + a \quad 19.5$$

donde *n* es el número de nodos (módulos) y a es el número de arcos (líneas de control). Para la arquitectura mostrada en la Figura 19.5,

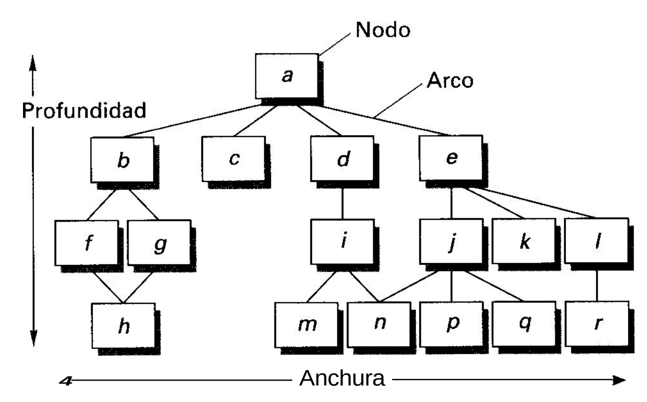


FIGURA 19.5. Métricas de morfología

$$tamaño = 17 + 18 = 35$$

profundidad= el camino más largo desde el nodo raíz (parte más alta) a un nodo hoja. Para la arquitectura mostrada en la Figura 19.5, la profundidad = **4.**

anchura = máximo número de nodos de cualquier nivel de la arquitectura. Para la arquitectura mostrada en la Figura 19.5, la anchura = 6.

Relación arco-a-nodo, $r = a / n$.

que mide la densidad de conectividad de la arquitectura y puede proporcionar una sencilla indicación del acoplamiento de la arquitectura. Para la arquitectura mostrada en la Figura 19.5, $r = 18 / 17 = 1{,}06$.

Cita:

La medición puede ser vista como un rodeo. Este rodeo es necesario porque los humanos muchas veces no estamos capacitados para tomar decisiones con claridad y objetividad [sin un soporte cuantitativo].

Horst Zuse

19.4.2. Métricas de diseño a nivel de componentes

Las métricas de diseño a nivel de componentes se concentran en las características internas de los componentes del software e incluyen medidas de las «3Cs» —la cohesión, acoplamiento y complejidad del módulo—. Estas tres medidas pueden ayudar al desarrollador de software a juzgar la calidad de un diseño a nivel de los componentes.

Las métricas presentadas en esta sección son de caja blanca en el sentido de que requieren conocimiento del trabajo interno del módulo en cuestión. Las métricas de diseño de los componentes se pueden aplicar una vez que se ha desarrollado un diseño procedimental. También se pueden retrasar hasta tener disponible el código fuente.



Es posible calcular medidos de la independencia funcional, acoplamiento y cohesión de un componente y usarlos paro valorar la calidad y el diseño.

Métricas de cohesión. Bieman y Ott [BIE94] definen una colección de métricas que proporcionan una indicación de la cohesión (Capítulo **13)** de un módulo. Las métricas se definen con cinco conceptos y medidas:

> ***Porción de datos.*** Dicho simplemente, una porción de datos es una marcha atrás a través de un módulo que busca valores de datos que afectan a la localización de módulo en el que empezó la marcha atrás. Debería resaltarse que se pueden definir tanto porciones de programas (que se centran en enunciados y condiciones) como porciones de datos.
>
> ***Muestras (tokens) de datos.*** Las variables definidas para un módulo pueden definirse como muestras de datos para el módulo.
>
> ***Señales de unión.*** El conjunto de muestras de datos que se encuentran en una o más porciones de datos.
>
> ***Señales de superunión.*** La muestras de datos comunes a todas las porciones de datos de un módulo.
>
> ***Pegajosidad.*** La pegajosidad relativa de una muestra de unión es directamente proporcional al número de porciones de datos que liga.

Bieman y Ott desarrollan métricas para *cohesiones funcionales fuertes* (CFF), *cohesiones funcionales débiles* (CFD) y *pegajosidad* (el grado relativo con el que las señales de unión ligan juntas porciones de datos). Estas métricas se pueden interpretar de la siguiente manera [BIE94]:

> Todas estas métricas de cohesión tienen valores que van de 0 a 1. Tienen un valor de 0 cuando un procedimiento tiene más de una salida y no muestra ningún atributo de cohesión indicado por una métrica particular. Un procedimiento sin muestras de superunión, sin muestras comunes a todas las porciones de datos, no tiene una cohesión funcional fuerte (no hay señales de datos que contribuyan a todas las salidas). Un procedimiento sin muestras de unión, es decir, sin muestras comunes a más de una porción de datos (en procedimientos con más de una porción de datos), no muestra una cohesión funcional débil y ninguna adhesividad (no hay señales de datos que contribuyan a más de una salida).

La cohesión funcional fuerte y la pegajosidad se obtienen cuando las métricas de Bieman y Ott toman un valor máximo de **1**.

Se deja un estudio más detallado de las métricas de Bieman y Ott para que sean revisadas sus fuentes [BIE94]. Sin embargo, para ilustrar el carácter de estas métricas, considere la métrica para la cohesión funcional fuerte:

$$\text{CFF}(i) = \text{SU}\ (\text{SA}(i)) \ / \ \text{muestra}(i) \qquad (19.6)$$

donde SU (SA(i)) denota muestra de superunión (el conjunto de señales de datos que se encuentran en todas las porciones de datos de un módulo i). Como la relación de muestras de superunión con respecto al número total de muestras en un módulo i aumenta hasta un valor máximo de 1, la cohesión funcional del módulo también aumenta.

Métricas de acoplamiento. El acoplamiento de módulo proporciona una indicación de la «conectividad» de un módulo con otros módulos, datos globales y el entorno exterior. En el Capítulo **13,** el acoplamiento se estudió en términos cualitativos.

Dhama [DHA95] ha propuesto una métrica para el acoplamiento del módulo que combina el acoplamiento de flujo de datos y de control, acoplamiento global y acoplamiento de entorno. Las medidas necesarias para calcular el acoplamiento de módulo se definen en términos de cada uno de los tres tipos de acoplamiento apuntados anteriormente.

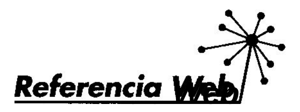
Para el acoplamiento de flujo de datos y de control:

d_i = número de parámetros de datos de entrada

c_i = número de parámetros de control de entrada

d_o = número de parámetros de datos de salida

c_o = número de parámetros de control de salida



El documento, «Sistema de métricas del software para el acoplamiento de módulos» podéis bajarlo de **www.isse.gmu.edu/faculty/ofut/rsrch/abstracts/mj-coupling.html**

Para el acoplamiento global:

g_d = número de variables globales usadas como datos

g_c = número de variables globales usadas como control

Para el acoplamiento de entorno:

w = número de módulos llamados (expansión)

r = número de módulos que llaman al módulo en cuestión (concentración)

Usando estas medidas, se define un indicador de acoplamiento de módulo, m_c, de la siguiente manera:

$$m_c = k / M$$

donde **k=1** es una constante de proporcionalidad[8].

$$M = d_i + a \times c_i + d_o + b \times c_o + g_d + c \times g_c + w + r$$

donde $a = b = c = 2.$

Cuanto mayor es el valor de m_c, menor es el acoplamiento de módulo. Por ejemplo, si un módulo tiene un solo parámetro de entrada y salida de datos, no accede a datos globales y es llamado por un solo módulo:

$$m_c = 1 / (1 + 0 + 1 + 0 + 0 + 0 + 1 + 0) = 1 / 3 = 0{,}33.$$

Deberíamos esperar que un módulo como éste presentara un acoplamiento bajo. De ahí que, un valor de $m_c = 0{,}33$

[8] El autor [DHA95] advierte que los valores de k y a, b y c (tratados en la siguiente ecuación) pueden ajustarse a medida que se van verificando experimentalmente.

implica un acoplamiento bajo. Por el contrario, si un módulo tiene 5 parámetros de salida y 5 parámetros de entrada de datos, un número igual de parámetros de control, accede a 10 elementos de datos globales y tiene una concentración de *3* y una expansión de 4,

$$m_c = 1/(5 + 2 \times 5 + 5 + 2 \times 5 + 10 + 0 + 3 + 4) = 0{,}02$$

el acoplamiento implicado es grande.

Para que aumente la métrica de acoplamiento a medida que aumenta el grado de acoplamiento, se puede definir una métrica de acoplamiento revisada como:

$$C = 1 - m_c$$

donde el grado de acoplamiento no aumenta linealmente **entre** un valor mínimo en el rango de 0,66 hasta un máximo que se aproxima a 1,0.

Métricas de complejidad. Se pueden calcular una variedad de métricas del software para determinar la complejidad del flujo de control del programa. Muchas de éstas se basan en una representación denominada grafo de flujo. Tal y como se dijo en el Capítulo 17, un grafo *es* una representación compuesta de nodos y enlaces (también denominados aristas). Cuando se dirigen los enlaces (aristas), el grafo de flujo es un grafo dirigido.

McCabe [MCC94] identifica un número importante de usos para las métricas de complejidad:

> Las métricas de complejidad pueden emplearse para predecir la información crítica sobre la fiabilidad y mantenimiento de sistemas software de análisis automáticos de código fuente (o información de diseño procedimental). Las métricas de complejidad también realimentan la información durante el proyecto de software para ayudar a controlar la [actividad del diseño]. Durante las pruebas y el mantenimiento, proporcionan una detallada información sobre los módulos software para ayudar a resaltar las áreas de inestabilidad potencial.

La métrica de complejidad más ampliamente usada (**y** debatida) para el software es la complejidad *ciclomática*, originalmente desarrollada por Thomas McCabe [MCC76] **y** estudiando con detalle en la Sección 17.4.2.

La métrica de McCabe proporciona una medida cuantitativa para probar la dificultad y una indicación de la fiabilidad última. Estudios experimentales indican una fuerte correlación entre la métrica de McCabe y el número de errores que existen en el código fuente, así como el tiempo requerido para encontrar y corregir dichos errores.



la complejidad ciclomática es solamente una más de los muchos métricas de complejidad.

McCabe también defiende que la complejidad ciclomática puede emplearse para proporcionar una indicación cuantitativa del tamaño máximo del módulo. Recogiendo datos de varios proyectos de programación reales, ha averiguado que una complejidad ciclomática de 10 parece ser el límite práctico superior para el tamaño de un módulo. Cuando la complejidad ciclomática de los módulos excedían ese valor, resultaba extremadamente difícil probar adecuadamente el módulo. Vea en el Capítulo 17 un estudio sobre la complejidad ciclomática como guía para el diseño de casos de prueba de caja blanca.

19.4.3. Métricas de diseño de interfaz

Aunque existe una significativa cantidad de literatura sobre el diseño de interfaces hombre-máquina (vea el Capítulo 15), se ha publicado relativamente poca información sobre métricas que proporcionen una visión interna de la calidad y facilidad de empleo de la interfaz.

Sears [SEA93] sugiere la conveniencia de la representación (CR) como una valiosa métrica de diseño para interfaces hombre-máquina. Una IGU (Interfaz Gráfica de Usuario) típica usa entidades de representación —iconos gráficos, texto, menús, ventanas y otras— para ayudar al usuario a completar tareas. Para realizar una tarea dada usando una IGU, el usuario debe moverse de una entidad de representación a otra. Las posiciones absolutas y relativas de cada entidad de representación, la frecuencia con que se utilizan y el «coste» de la transición de una entidad de representación a la siguiente contribuirán a la conveniencia de la interfaz.

Cita:
La belleza —el ajuste de todas las partes que hace que no se pueda añadir, quitar o cambiar algo sin impactar en la armonía del conjunto—.
León Alberti (1404 – 1472).

Para una representación específica (por ejemplo, un diseño de una IGU específica), se pueden asignar costes a cada secuencia de acciones de acuerdo con la siguiente relación:

$$\text{Costes} = \sum [\text{frecuencia de transición } (k) \times \text{coste de transición } (k)] \quad (19.7)$$

donde ***k*** es la transición específica de una entidad de representación a la siguiente cuando se realiza una tarea específica. Esta suma se da con todas las transiciones de una tarea en particular o conjunto de tareas requeridas para conseguir alguna función de la aplicación. El coste puede estar caracterizado en términos de tiempo, retraso del proceso o cualquier otro valor razonable, tal como la distancia que debe moverse el ratón entre entidades de la representación.

La conveniencia de la representación se define como:

$$CR = 100 \times [(\text{coste de la representación Óptima CR}) / (\text{coste de la representación propuesta})] \quad (19.8)$$

donde CR = 100 para una representación Óptima.

Para calcular la representación óptima de una IGU, la superficie de la interfaz (el área de la pantalla) se divide en una cuadrícula. Cada cuadro de la cuadncula representa una posible posición de una entidad de la representación. Para una cuadrícula con *N* posibles posi-

ciones y K diferentes entidades de representación para colocar, el número posible de distribuciones se representa de la siguiente manera [SEA93]:

número posible de distribuciones =

$$= [N!/(K! \times (N-K)!] \times K! \qquad (19.9)$$

A medida que crece el número de posiciones de representación, el número de distribuciones posibles se hace muy grande. Para encontrar la representación óptima (menor coste). Sears [SEA93] propone un algoritmo de búsqueda en árbol.



*las métricas del diseno de interface son **válidas,** pero sobre todo, son absolutamente necesarios para **asegurar** que a los usuarios finales les **agrada** la **inteface** y estén **satisfechos** con los **interacciones** requeridas.*

La **CR** se emplea para valorar diferentes distribuciones propuestas de IGU y la sensibilidad de una representación en particular a los cambios en las descripciones de tareas (por ejemplo, cambios en la secuencia y/o frecuencia de transiciones). El diseñador de interfaces puede emplear el cambio en la conveniencia de la representación, **ACR,** como guía en la elección de la mejor representación de IGU para una aplicación en particular.

Es importante apuntar que la selección de un diseño de IGU puede guiarse con métricas tales como **CR,** pero el árbitro final debería ser la respuesta del usuario basada en prototipos de IGU. Nielsen y Levy [NIE94] informan que «existe una gran posibilidad de éxito si se elige una interfaz basándose solamente en la opinión del usuario. El rendimiento medio de tareas de usuario y su satisfacción con la IGU están altamente relacionadas.»

19.5 MÉTRICAS DEL CÓDIGO FUENTE

La teoría de Halstead de la ciencia del software [HAL77] es «probablemente la mejor conocida y más minuciosamente estudiada... medidas compuestas de la complejidad (software)» **[CURSO].** La ciencia del software propone las primeras «leyes» analíticas para el software de computadora[9].



El cerebro humano sigue un conjunto rígido de reglas que conoce [en el desarrollo de algoritmos].
Maurice Halstead

La ciencia del software asigna leyes cuantitativas al desarrollo del software de computadora, usando un conjunto de medidas primitivas que pueden obtenerse una vez que se ha generado o estimado el código después de completar el diseño. Estas medidas se listan a continuación.

n_1: el número de operadores diferentes que aparecen en el programa

n_2: el número de operandos diferentes que aparecen en el programa

N_1: el número total de veces que aparece el operador

N_2: el número total de veces que aparece el operando



*los **operadores** incluyen todos las **construcciones** del **flujo** de **control,** condiciones y **operaciones matemáticas.** los **operandos agrupan** todos las **variables** y constantes del programo.*

Halstead usa las medidas primitivas para desarrollar expresiones para la *longitud* global del programa; *volumen mínimo potencial* para un algoritmo; el *volumen real* (número de bits requeridos para especificar un programa); *el nivel del programa* (una medida de la complejidad del software); *nivel del lenguaje* (una constante para un lenguaje dado); y otras características tales como esfuerzo de desarrollo, tiempo de desarrollo e incluso el número esperado de fallos en el software.

Halstead expone que la longitud N se puede estimar como:

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2 \qquad (19.10)$$

y el volumen de programa se puede definir como:

$$V = N \log_2 (n_1 + n_2) \qquad (19.11)$$

Se debería tener en cuenta que V variará con el lenguaje de programación y representa el volumen de información (en bits) necesarios para especificar un programa.

Teóricamente, debe existir un volumen mínimo para un algoritmo. Halstead define una relación de volumen L como la relación de volumen de la forma más compacta de un programa con respecto al volumen real del programa. Por tanto, L debe ser siempre menor de uno. En términos de medidas primitivas, la relación de volumen se puede expresar como

$$L = 2/n_1 \times n_2/N_2 \qquad (19.12)$$

[9] Se debería resaltar que las «leyes» de Halstead han generado una sustancial controversia y que no todo el mundo está de acuerdo con que la teoría subyacente sea correcta. Sin embargo, se han realizado verificaciones experimentales de las averiguaciones de Halstead para varios lenguajes de programación (por ejemplo, [FEL89]).

El trabajo de Halstead se presta a la verificación experimental y de hecho se ha desarrollado una gran labor de investigación en la ciencia del software. Un estudio de este trabajo estaría fuera del alcance de este libro, pero puede decirse que se ha llegado a un buen acuerdo entre lo previsto analíticamente y los resultados experimentales. Para más información, vea [ZUS90], [FEN91] y [ZUS97].

19.6 MÉTRICAS PARA PRUEBAS

Aunque se ha escrito mucho sobre las métricas del software para pruebas (por ejemplo, [HET93]), la mayoría de las métricas propuestas se concentran en el proceso de prueba, no en las características técnicas de las pruebas mismas. En general, los responsables de las pruebas deben fiarse de las métricas de análisis, diseño y código para que les guíen en el diseño y ejecución de los casos de prueba.



Los métricos de lo prueba desembocan en los siguientes categorías: (1) métricas que ayudan o determinar el número de pruebas requeridos en los distintos niveles de la prueba; (2) métricas poro cubrir la pruebo de un componente dado.

Las métricas basadas en la función (Sección 19.3.1) pueden emplearse para predecir el esfuerzo global de las pruebas. Se pueden juntar y correlacionar varias características a nivel de proyecto (por ejemplo, esfuerzo y tiempo para las pruebas, errores descubiertos, número de casos de prueba producidos) de proyectos anteriores con el número de PF producidos por un equipo del proyecto. El equipo puede después predecir los valores «esperados» de estas características del proyecto actual.

La métrica *bang* puede proporcionar una indicación del número de casos de prueba necesarios para examinar las medidas primitivas tratadas en la sección 19.3.2. El número de primitivas funcionales (PFu), elementos de datos (DE), objetos (OB), relaciones (RE), estados **(ES) y** transiciones (TR) pueden emplearse para predecir el número y tipos de prueba del software de caja negra y de caja blanca. Por ejemplo, el número de pruebas asociadas con la interfaz hombre-máquina se puede estimar examinando: (1) el número de transiciones (TR) contenidas en la representación estado-transición del IHM y evaluando las pruebas requeridas para ejecutar cada transición, (2) el número de objetos de datos (OB) que se mueven a través de la interfaz y (3) el número de elementos de datos que se introducen o salen.

Las métricas del diseño arquitectónico proporcionan información sobre la facilidad o dificultad asociada con la prueba de integración (Capítulo 18) y la necesidad de software especializado de prueba (por ejemplo, matrices y controladores). La complejidad ciclomática (una métrica de diseño de componentes) se encuentra en el núcleo de las pruebas de caminos básicos, un método de diseño de casos de prueba presentado en el Capítulo 17. Además, la complejidad ciclomática puede usarse para elegir módulos como candidatos para pruebas más profundas (Capítulo 18). Los módulos con gran complejidad ciclomática tienen más probabilidad de tendencia a errores que los módulos con menor complejidad ciclomática.

Por esta razón, el analista debería invertir un esfuerzo extra para descubrir errores en el módulo antes de integrarlo en un sistema. Es esfuerzo de las pruebas también se puede estimar usando métricas obtenidas de medidas de Halstead (Sección 19.5). Usando la definición del volumen de un programa, V, y nivel de programa, NP, el esfuerzo de la ciencia del software, e, puede calcularse como:

$$NP = 1 / [(n_1/2) \times (N_2/n_2)] \quad (19.13a)$$

$$e = V / NP \quad (19.13b)$$

El porcentaje del esfuerzo global de pruebas a asignar a un módulo **k** se puede estimar usando la siguiente relación:

$$\text{porcentaje de esfuerzo de pruebas (k)} = e(k)/\sum e(i) \quad (19.14)$$

donde *e(k)* se calcula para el módulo k usando las ecuaciones (19.13) y la suma en el denominador de la ecuación (19.14) es la suma del esfuerzo de la ciencia del software a lo largo de todos los módulos del sistema.

A medida que se van haciendo las pruebas, tres medidas diferentes proporcionan una indicación de la compleción de las pruebas. Una medida de la *amplitud de las pruebas* proporciona una indicación de cuantos requisitos (del número total de ellos) se han probado. Esto proporciona una indicación de la compleción del plan de pruebas. La profundidad de las pruebas es una medida del porcentaje de los caminos básicos independientes probados en relación al número total de estos caminos en el programa. Se puede calcular una estimación razonablemente exacta del número de caminos básicos sumando la complejidad ciclomática de todos los módulos del programa. Finalmente, a medida que se van haciendo las pruebas y se recogen los datos de los errores, se pueden emplear los perfiles *de fallos* para dar prioridad y categorizar los errores descubiertos. La prioridad indica la severidad del problema. Las categorías de los fallos proporcionan una descripción de un error, de manera que se puedan llevar a cabo análisis estadísticos de errores.

19.7 MÉTRICAS DEL MANTENIMIENTO

Todas las métricas del software presentadas en este capítulo pueden usarse para el desarrollo de nuevo software y para el mantenimiento del existente. Sin embargo, se han propuesto métricas diseñadas explícitamente para actividades de mantenimiento.

El estándar E E E 982.1-1988 [IEE94] sugiere un índice de madurez del software (IMS) que proporciona una indicación de la estabilidad de un producto software (basada en los cambios que ocurren con cada versión del producto). Se determina la siguiente información:

M_T = número de módulos en la versión actual

F_c = número de módulos en **la** versión actual que se han cambiado

F_a = número de módulos en la versión actual que se han añadido

F_d = número de módulos de la versión anterior **que** se han borrado en **la** versión actual

El índice de madurez del software se calcula de la siguiente manera:

$$IMS = [M_T - (F_a + F_c + F_d)] / M_T \qquad (19.15)$$

A medida que el IMS se aproxima a **1**,0 el producto se empieza a estabilizar. EL IMS puede emplearse también como métrica para la planificación de las actividades de mantenimiento del software. El tiempo medio para producir una versión de un producto software puede correlacionarse con el IMS desarrollándose modelos empíricos para el mantenimiento.

RESUMEN

Las métricas del software proporcionan una manera cuantitativa de valorar la calidad de los atributos internos del producto, permitiendo por tanto al ingeniero valorar la calidad antes de construir el producto. Las métricas proporcionan la visión interna necesaria para crear modelos efectivos de análisis y diseño, un código sólido y pruebas minuciosas.

Para que sea útil en el contexto del mundo real, una métrica del software debe ser simple y calculable, persuasiva, consistente y objetiva. Debería ser independiente del lenguaje de programación y proporcionar una realimentación eficaz para el desarrollador de software.

Las métricas del modelo de análisis se concentran en la función, los datos y el comportamiento (los tres componentes del modelo de análisis). El punto de función **y** la métrica *bang* proporcionan medidas cuantitativas para evaluar el modelo de análisis. Las métricas del diseño consideran aspectos de alto nivel, del nivel de componentes y de diseño de interfaz. Las métricas de diseño de alto nivel consideran los aspectos arquitectónicos y estructurales del modelo de diseño. Las métricas de diseño de nivel de componentes proporcionan una indicación de la calidad del módulo estableciendo medidas indirectas de la cohesión, acoplamiento y complejidad. Las métricas de diseño de interfaz proporcionan una indicación de la conveniencia de la representación de una IGU.

La ciencia del software proporciona un intrigante conjunto de métricas a nivel de código fuente. Usando el número de operadores y operandos presentes en el código, la ciencia del software proporciona una variedad de métricas que pueden usarse para valorar la calidad del programa.

Se han propuesto pocas métricas técnicas para un empleo directo en las pruebas y mantenimiento del software. Sin embargo, se pueden emplear muchas otras métricas técnicas para guiar el proceso de las pruebas y como mecanismo para valorar la facilidad de mantenimiento de un programa de computadora.

REFERENCIAS

[BAS84] Basili, y V.R. D.M. Weiss, «A Methodology for Collecting Valid Software Engineering Data», *IEEE Trans. Software Engineering,* vol. SE-10, 1984, **pp.** 728-738.

[BIE94] Bieman, J.M., y L.M. Ott, «Measuring Functional Cohesion», *IEEE Trans. Software Engineering,* vol. *20,* n.º 8, Agosto 1994, **pp.** 308-320.

[BRI96] Briand, L.C., S. Morasca, y V.R. Basili, «Property-Based Software Engineering Measurement», ***IEEE*** *Trans. Software Engineering,* vol *22,* n.º 1, Enero 1996, **pp.** 68-85.

[CAR90] Card, D., y N. R. L. Glass, *Measuring Software Design Quality,* Prentice-Hall, 1990.

[CAV78] Cavano, J.P., y J.A. McCall, «A Framework for the Measurement of Software Quality», *Proc. ACM Software Quality Assurance Wokshop,* Noviembre 1978, pp. 133-139.

[CHA89] Charette, R.N., Software Engineering Risk Analysis and Management, McGraw-Hill/Intertext, 1989.

[CURSO] Curtis, W., «Management and Experimentation in Software Engineering», *Proc. IEEE,* vol. 68, n.º 9, Septiembre 1980.

[DAV93] Davis, A., et al., «Identifying and Measuring Quality in a Software Requirements Specification», Proc. 1st

Intl. Software Metrics Symposium, IEEE, Baltimore, MD, May 1993, pp. 141-152.

[DEM81] DeMillo, R.A., y R.J. Lipton, «Software Project Forecasting», *Software Metrics* (A.J. Perlis, F.G. Sayward y M. Shaw, ed.), MIT Press, 1981, pp. 77-89.

[DEM82] DeMarco, T., *Controlling Software Projects*, Yourdon Press, 1982.

[DHA95] Dhama, H., «Quantitative Models of Cohesion and Coupling in Software», *Journal of Systems and Software*, vol. 29, n.º 4, Abril 1995.

[EJI91] Ejiogu, L., *Software Engineering with Formal Metrics*, QED Publishing, 1991.

[FEL89] Felican, L., y G. Zalateu, «Validating Halstead's Theory for Pascal Programs», IEEE Trans. Software Engineering, vol. 15, n.º 2, Diciembre 1989, pp. 1630-1632.

[FEN91] Fenton, N., Software Metrics, Chapman & Hall, 1991.

[FEN94] Fenton, N., «Software Measurement: A Necessary Scientific Basis», *IEEE Trans. Software Engineering*, vol. *20*, n.º 3, Marzo 1994, pp. 199-206.

[GRA87] Grady, R. B., y D.L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, 1987.

[HA77] Halstead, M., *Elements of Software Science*, N.ºrth Holland, 1977.

[HEN81] Henry, S., y D. Kafura, «Software Structure Metrics Based on information Flow», *IEEE Trans. Software Engineering*, vol. SE-7, n.º 5, Septiembre 1981, pp. 510-518.

[HET93] Hetzel, B., *Making Software Measurement Work*, QED Publishing, 1993.

[IEE94] *Software Engineering Standards*, 1994, IEEE, 1994.

[KYB84] Kyburg, H.E., *Theory and Measurement*, Cambridge University Press, 1984.

[MCC76] McCabe, T.J., «A Software Complexity Measure», IEEE Trans. Software Engineering, vol. 2, Diciembre 1976, pp. 308-320.

[MCC77] McCall, J., P. Richards, y G. Walters, «Factors in Software Quality», 3 vols., NTIS AD-A049-014, 015, 055, Noviembre 1977.

[MCC89] McCabe, T.J., y C.W. Butler, «Design Complexity Measurement and Testing», CACM, vol. 32, n.º 12, Diciembre 1989, pp. 1415-1425.

[MCC94] McCabe, T.J., y A.H. Watson, «Software Complexity», Crosstalk, vol. 7, n.º 12, Diciembre 1994, pp. 5-9.

[NIE94] Nielsen, J., y J. Levy, «Measuring Usability: Preference vs. Performance», CACM, vol. 37, n.º 4, Abril 1994, pp. 76-85.

[ROC94] Roche, J.M., «Software Metrics and Measurement Principles», *Software Engineering Notes*, ACM, vol. 19, n.º 1, Enero 1994, pp. 76-85.

[SEA93] Sears, A., «Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout», *IEEE Trans. Software Engineering*, vol. 19, n.º 7, Julio 1993, pp. 707-719.

[USA87] *Management Quality Insight*, AFCSP 800-14 **(U.S.** Air Force), 20 Enero, 1987.

[ZUS90] Zuse, H., *Software Complexity: Measures and Methods*, DeGruyter, Nueva York, 1990.

[ZUS97] Zuse, H., *A Framework of Software Measurement*, DeGruyter, Nueva York, 1997.

PROBLEMAS Y PUNTOS A CONSIDERAR

19.1. La teoría de la medición es un tema avanzado que tiene una gran influencia en las métricas del software. Mediante [FEN91], [ZUS90] y [KYB84] u otras fuentes, escriba una pequeña redacción que recoja los principales principios de la teoría de la medición. Proyecto individual: Prepare una conferencia sobre el tema y expóngala en clase.

19.2. Los factores de calidad de McCall se desarrollaron durante los años setenta. Casi todos los aspectos de la informática han cambiado dramáticamente desde que se desarrollaron, y sin embargo, los factores de McCall todavía se aplican en el software moderno. ¿Puede sacar alguna conclusión basada en este hecho?

19.3. Por qué no se puede desarrollar una única métrica que lo abarque todo para la complejidad o calidad de un programa?

19.4. Revise el modelo de análisis que desarrolló como parte del Problema 12.13. Mediante las directrices de la Sección 19.3.1., desarrolle una estimación del número de puntos función asociados con SSRB.

19.5. Revise el modelo de análisis que desarrolló como parte del Problema 12.13. Mediante las directrices de la Sección 19.3.2., desarrolle cuentas primitivas para la métrica bang. ¿El sistema SSRB es de dominio de función o de datos?

19.6. Calcule el valor de la métrica bang usando las medidas que desarrolló en el Problema 19.5.

19.7. Cree un modelo de diseño completo para un sistema propuesto por su profesor. Calcule la complejidad estructural y de datos usando las métricas descritas en la Sección 19.4.1. Calcule también las métricas de Henry-Kafura y de morfología para el modelo de diseño.

19.8. Un sistema de información grande tiene 1.140 módulos. Hay 96 módulos que realizan funciones de control y coordinación, y 490 cuya función depende de un proceso anterior. El sistema procesa aproximadamente 220 objetos de datos con una media de tres atributos por objeto de datos. Hay 140 elementos de la base de datos Únicos y 90 segmentos de base de datos diferentes. 600 módulos tienen un solo punto de entrada y de salida. Calcule el ICDE del sistema.

19.9. Investigue el trabajo de Bieman y Ott [BIE94] y desarrolle un ejemplo completo que ilustre el cálculo de su métrica de cohesión. Asegúrese de indicar cómo se determinan las porciones de datos, señales de datos y señales de unión y superunión.

19.10. Seleccione cinco módulos existentes de un programa de computadora. Mediante la métrica de Dhama descrita en la Sección 19.4.2., calcule el valor de acoplamiento de cada módulo.

19.11. Desarrolle una herramienta de software que calcule la complejidad ciclomática de un módulo de lenguaje de programación. Elija el lenguaje.

19.12. Desarrolle una herramienta de software que calcule la conveniencia de la representación de una IGU. Esa herramienta debería permitirle asignar el coste de transición entre las entidades de la representación. (Nota: fíjese en que el tamaño potencial del número de las altemativas de representación crece mucho a medida que aumenta el número de posibles posiciones de cuadrícula.)

19.13. Desarrolle una pequeña herramienta de software que realice un análisis Halstead de un código fuente de un lenguaje de programación a su elección.

19.14. Haga una investigación y escriba un documento sobre la relación entre las métricas de la calidad del software de Halstead y la de McCabe (con respecto a la cuenta de errores). ¿Son convincentes los datos? Recomiende unas directrices para la aplicación de estas métricas.

19.15. Investigue documentos recientes en busca de métricas específicamente diseñadas para el diseño de casos de prueba. Exponga los resultados en clase.

19.16. Un sistema heredado tiene 940 módulos. La última versión requiere el cambio de 90 de estos módulos. Además, se añaden 40 módulos nuevos y se quitaron 12 módulos antiguos. Calcule el índice de madurez del software del sistema.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Sorprendentemente hay un gran número de libros dedicados a las métricas del software, aunque la mayoría están enfocados a métricas de proceso y proyecto excluyendo las métricas técnicas. Zuse [ZUS97] ha escrito el más completo tratado de métricas técnicas publicado hasta la fecha.

Los libros de Card y Glass [CAR90], Zuse [ZUS90], Fenton [FEN91], Ejiogu [EJI91], Moeller y Paulish (Métricas del Software, Chapman & Hall, 1993) y Hetzel [HET93] son referencias sobre las métricas técnicas en detalle. Además, merece la pena examinar los siguientes libros:

Conte, S.D., H.E. Dunsmore, y V.Y. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings, 1984.

Fenton, N.E., y S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2.ª ed., PWS Publishing Co., 1998.

Grady, R.B. *Practica1 Software Metrics for Project Mangement and Process Improvement*, Prentice Hall, 1992.

Perlis, A., et al., *Software Metrics: An Analysis and Evaluation*, MIT Press, 1981.

Sheppard, M., *Software Engineering Metrics*, McGraw-Hill, 1992.

La teoría de la medición del software la presentan Denvir, Herman, y Whitty en una colección de documentos *(Proceedings of the International BCS-FACS Workshop: Formal Aspects of Measurement*, Springer-Verlag, 1992). Shepperd *(Foundations of Software Measurement*, Prentice Hall, 1996) también tratan la teoría de la medición en detalle.

Una relación de docenas de usos de las métricas del software están presentadas en [IEE94]. En general, una discusión de cada métrica ha sido reducida a lo esencial «las primitivas» (medidas) requeridas para calcular las métricas y las adecuadas relaciones a efectos de los cálculos. Un apéndice del libro suministra más información y referencias sobre el tema.

Una amplia variedad de fuentes de información sobre métricas técnicas y elementos relacionados están disponibles en Intemet. Una lista actualizada de referencias de páginas web sobre métricas técnicas la puedes encontrar en **http://www.pressman5.com.**

PARTE

IV

INGENIERÍA DEL SOFTWARE ORIENTADA A OBJETOS

EN esta parte de *Ingeniería del software: un enfoque práctico* consideramos los conceptos técnicos, métodos y medidas aplicables al análisis, diseño y pruebas de software orientado a objetos. En los siguientes capítulos responderemos las siguientes preguntas:

- ¿Cuáles son los conceptos y principios básicos aplicables al pensamiento orientado a objetos?
- ¿En qué se diferencian el enfoque tradicional y el orientado a objetos?
- ¿Cómo deben gestionarse y planificarse los proyectos orientados objetos?
- ¿Qué es el análisis orientado a objetos y cómo permiten sus distintos modelos comprender las clases, sus relaciones y comportamiento al ingeniero del software?
- ¿Cuáles son los elementos de un modelo de diseño orientado a objetos?
- ¿Cuáles son los conceptos y principios básicos aplicables a las pruebas de software orientado a objetos?
- ¿Cómo cambian las estrategias de prueba y los métodos de diseño de casos de prueba cuando se considera el software orientado a objetos?
- ¿Qué métricas técnicas están disponibles para evaluar la calidad del software orientado a objetos?

Una vez contestadas estas preguntas, comprenderá cómo analizar, diseñar, implementar y probar el software usando el paradigma de orientación a objetos.

CAPÍTULO

20 CONCEPTOS Y PRINCIPIOS ORIENTADOS A OBJETOS

VIVIMOS en un mundo de objetos. Estos objetos existen en la naturaleza, en entidades hechas por el hombre, en los negocios y en los productos que usamos. Pueden ser clasificados, descritos, organizados, combinados, manipulados y creados. Por esto no es sorprendente que se proponga una visión orientada a objetos para la creación de software de computadora, una abstracción que modela el mundo de forma tal que nos ayuda a entenderlo y gobernarlo mejor.

La primera vez que se propuso un enfoque orientado a objetos para el desarrollo de software fue a finales de los años sesenta. Sin embargo, las tecnologías de objetos han necesitado casi veinte años para llegar a ser ampliamente usadas. Durante los años 90, la ingeniería del software orientada a objetos se convirtió en el paradigma de elección para muchos productores de software y para un creciente número de sistemas de información y profesionales de la ingeniería. A medida que pasa el tiempo, las tecnologías de objetos están sustituyendo a los enfoques clásicos de desarrollo de software. Una pregunta importante es: ¿Por qué?

La respuesta (como muchas otras respuestas a interrogantes sobre ingeniería del software) no es sencilla. Algunas personas argumentarían que los profesionales del software sencillamente añoraban un nuevo enfoque, pero esta visión es muy simplista. Las tecnologías de objeto llevan un número de beneficios inherentes que proporcionan ventajas a los niveles de dirección y técnico.

VISTAZO RÁPIDO

¿Qué es? Hay muchas formas de enfocar un problema utilizando una solución basada en el software. Un enfoque muy utilizado es la visión orientada a objetos. El dominio del problema se caracteriza mediante un conjunto de objetos con atributos y comportamientos específicos. Los objetos son manipulados mediante una colección de funciones (llamadas métodos, operaciones o servicios) y se comunican entre ellos mediante un protocolo de mensajes. Los objetos se clasifican mediante clases y subclases.

¿Quién lo hace? La definición de objetos implica la descripción de atributos, comportamientos, operaciones y mensajes. Esta actividad la realiza un ingeniero del software.

¿Por qué es importante? Un objeto encapsula tanto datos como los procesos que se aplican a esos datos. Esta importante característica permite construir clases de objetos e inherentemente construir bibliotecas de objetos y clases reutilizables. El paradigma de orientación a objetos es tan atractivo para tantas organizaciones de desarrollo de software debido a que la reutilización es un atributo importantísimo en la ingeniería. del software. Además, los componentes de software derivados mediante el paradigma de objetos muestran características (como la independencia funcional, la ocultación de información, etc.) asociadas con el software de alta calidad.

¿Cuáles son los pasos? La ingeniería del software orientado a objetos sigue los mismos pasos que el enfoque convencional. El análisis identifica las clases y objetos relevantes en el dominio del problema, el diseño proporciona detalles sobre la arquitectura, las interfaces y los componentes, la implementación (utilizando un lenguaje orientado a objetos) transforma el diseño en código, y las pruebas chequean tanto la arquitectura como las interfaces y los componentes.

¿Cuál es el producto obtenido? Se produce un conjunto de modelos orientados a objetos. Estos modelos describen los requisitos, el diseño, el código y los procesos de pruebas para un sistema o producto.

¿Cómo puedo estar seguro de que lo he hecho correctamente? En cada etapa se revisa la claridad de los productos de trabajo orientados a objetos, su corrección, compleción y consistencia con los requisitos del cliente y entre ellos.

Las tecnologías de objetos llevan a reutilizar, y la reutilización (de componente de software) lleva a un desarrollo de software más rápido y a programas de mejor calidad. El software orientado a objetos es más fácil de mantener debido a que su estructura es inherentemente poco acoplada. Esto lleva a menores efectos colaterales cuando se deben hacer cambios y provoca menos frustración en el ingeniero del software y en el cliente. Además, los sistemas orientados a objetos son más fáciles de adaptar y más fácilmente escalables (por ejemplo: pueden crearse grandes sistemas ensamblando subsistemas reutilizables).

En este capítulo presentamos los principios y conceptos básicos que forman el fundamento para la comprensión de tecnologías de objetos.

20.1 EL PARADIGMA ORIENTADO A OBJETOS

Durante muchos años el término orientado a objetos (OO) se usó para referirse a un enfoque de desarrollo de software que usaba uno de los lenguajes orientados a objetos (Ada 95, C++, Eiffel, Smalltalk, etc.). Hoy en día el paradigma OO encierra una completa visión de la ingeniería del software. Edward Berard hace notar esto cuando declara [BER93]:

> Los beneficios de la tecnología orientada a objetos se fortalecen si se usa antes y durante el proceso de ingeniería del software. Esta tecnología orientada a objetos considerada debe hacer sentir **su** impacto en todo el proceso de ingeniería del software. Un simple uso de programación orientada a objetos (POO) no brindará los mejores resultados. Los ingenieros del software y **sus** directores deben considerar tales elementos el análisis de requisitos orientado a objetos (AROO), el diseño orientado a objetos (DOO), el análisis del dominio orientado a objetos (ADOO), sistemas de gestión de bases de dalos orientadas a objetos (SGBDOO) y la ingeniena del software orientada a objetos asistida por computadora (ISOOAC).



Con los objetos es realmente más fácil construir modelos [paro sistemas complejos] que dedicarse o la programación secuenciol
David Taylor

El lector familiarizado con el enfoque convencional de ingeniería del software (presentada en la tercera parte de este libro) puede reaccionar ante esta declaración con esta pregunta: «¿Cuál es la gran ventaja? Usamos el análisis, diseño, la programación, las pruebas y otras tecnologías relacionadas cuando realizamos ingeniería del software según métodos clásicos. ¿Por **qué** debe ser **la** OO diferente?» Ciertamente, ¿por qué debe ser la OO diferente? En pocas palabras, ¡no debiera!



los sistemas OO utilizan un modelo de ingeniería mediante proceso evolutivo. Más adelante, en este mismo capítulo, nos referiremos a él como el modelo recursivo paralelo.

En el Capítulo 2, examinamos un número de modelos de procesos diferentes para la ingeniería del software. Aunque ninguno de estos modelos pudo ser adaptado para su uso con **OO,** la mejor selección reconocería que los sistemas OO tienden a evolucionar con el tiempo. Por esto, un modelo evolutivo de proceso acoplado con un enfoque que fomenta el ensamblaje (reutilización) de componentes es el mejor paradigma para ingeniería del software OO. En la Figura 20.1 el modelo de proceso **de** ensamblaje de componentes (Capítulo 2) ha sido adaptado a la ingeniería del software OO.



Una de las listos más completas de recursos OO en la web puede consultarse en **mini.net/cetus/software.html.**

El proceso OO se mueve a través de una espiral evolutiva que comienza con la comunicación con el usuario. Es aquí donde se define el dominio del problema y se identifican las clases básicas del problema (tratadas más tarde en este capítulo). La planificación y el análisis de riesgos establecen una base para el plan del **proyecto** OO. El trabajo técnico asociado con la ingeniería del software OO sigue el camino iterativo mostrado en la caja sombreada. La ingeniería del software OO hace hincapié en la reutilización. Por lo tanto, las clases se

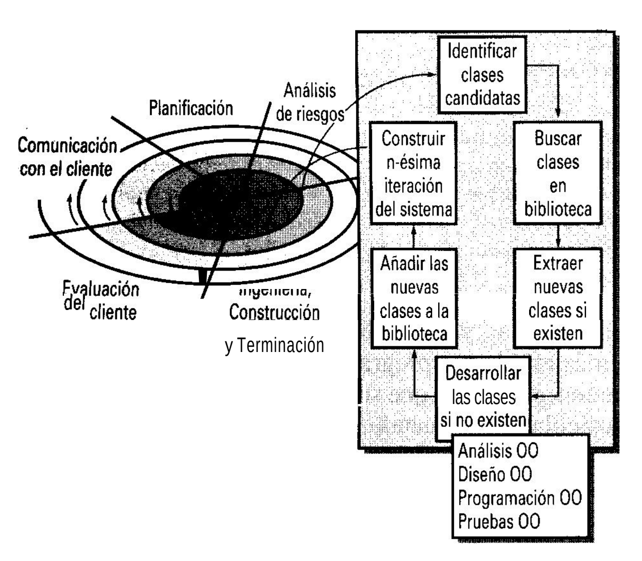


FIGURA 20.1. El modelo de proceso OO.

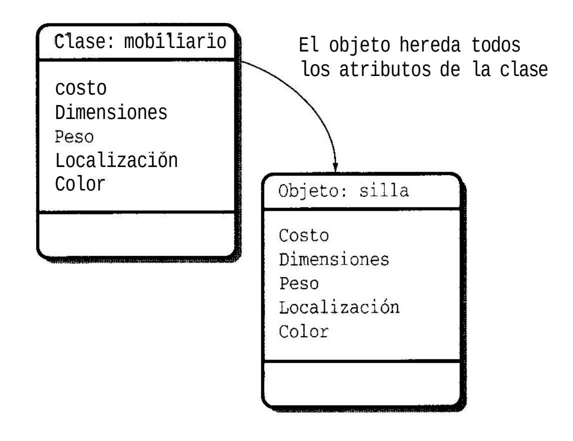


FIGURA 20.2. Herencia de clase a objeto.

buscan en una biblioteca (de clases OO existentes) antes de construirse. Cuando una clase no puede encontrarse en la biblioteca, el desarrollador de software aplica análisis orientado a objetos **(AOO),** diseño orientado a objetos (DOO), programación orientada a objetos (POO) y pruebas orientadas a objetos (PROO) para crear la clase y los objetos derivados de la clase. La nueva clase se pone en la biblioteca de tal manera que pueda ser reutilizada en el futuro.

La visión orientada a objetos demanda un enfoque evolutivo de la ingeniería del software. Como veremos a través de este y los próximos capítulos, será extremadamente difícil definir las clases necesarias para un gran sistema o producto en una sola iteracción. **A** medida que el análisis OO y los modelos de diseño evolucionan, se hace patente la necesidad de clases adicionales. Es por esta razón por lo que el paradigma arriba descrito funciona mejor para la OO.

20.2 CONCEPTOS DE ORIENTACIÓN A OBJETOS

Cualquier discusión sobre ingeniería del software orientada a objetos debe comenzar por el término *orientado **a** objetos.* ¿Qué es un punto de vista orientado a objetos? ¿Qué hace que un método sea considerado como orientado a objetos? ¿Qué es un objeto? Durante años han existido muchas opiniones diferentes (p. ej.: **[BER93], [TAY90], [STR88], [BOO86])** sobre las respuestas correctas a estas preguntas. En los párrafos siguientes trataremos de sintetizar las más comunes de éstas.



La programación orientada a objetos no es tanto una técnica de codificación de paquetes como una manera de que los constructores de software encapsulen funcionalidades para proporcionárselas **a sus clientes.**
Brad Cox

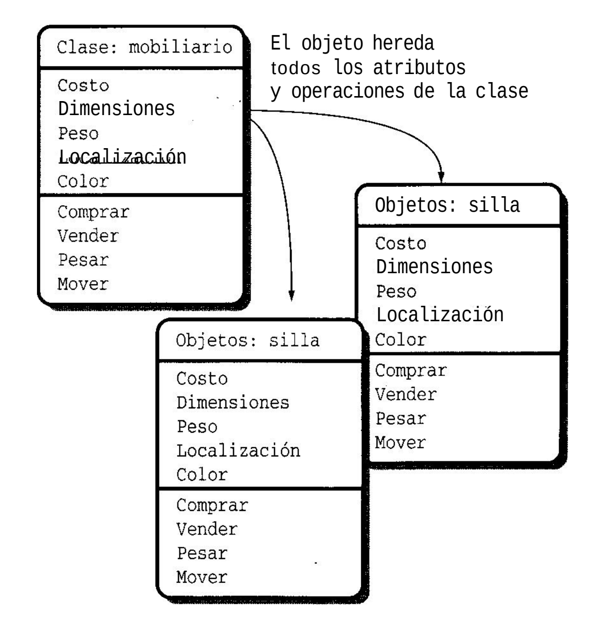


FIGURA 20.3. Herencia operaciones de clase a objeto.

Para entender la visión orientada a objetos, consideremos un ejemplo de un objeto del mundo real —la cosa sobre la que usted está sentado ahora mismo—, una silla. **Silla** es un miembro (el término *instancia* también se usa) de una clase mucho más grande de objetos que llamaremos **Mobiliario.** Un conjunto de atributos genéricos puede asociarse con cada objeto, en la clase **Mobiliario.** Por ejemplo, todo mueble tiene un costo, dimensiones, peso, localización y color, entre otros muchos posibles atributos. Estos son aplicables a cualquier elemento sobre el que se hable, una mesa o silla, un sofá o un armario. Como **Silla** es un miembro de la clase **Mobiliario,** hereda todos los atributos definidos para dicha clase. Este concepto se ilustra en la Figura 20.2 utilizando una notación conocida como UML. En dicha figura, la caja con una esquina doblada representa un comentario en un lenguaje de programación.

Una vez definida la clase, los atributos pueden reutilizarse al crear nuevas instancias de la clase. Por ejemplo, supongamos que debemos definir un nuevo objeto llamado **Sillesa** (un cruce entre una silla y una mesa) que es un miembro de la clase **Mobiliario.** La **Sillesa** hereda todos los atributos de **Mobiliario.**

Hemos intentado definir una clase describiendo sus atributos, pero algo falta. Todo objeto en la clase mobiliario puede manipularse de varias maneras. Puede comprarse y venderse, modificarse físicamente (por ejemplo, usted puede eliminar una pata o pintar el objeto de púrpura) o moverse de un lugar a otro. Cada una de estas *operaciones* (otros términos son *servicios* o *métodos)* modificará uno *o* más atributos del objeto. Por ejemplo, si el atributo **localización** es un dato compuesto definido como:

```
localización = edificio + piso + habitación
```

entonces una operación denominada *mover* modificaría uno *o* más de los elementos dato **(edificio, piso** o **habitación)** que conforman el atributo **localización.** Para hacer esto, *mover* debe tener «conocimiento» sobre estos elementos. La operación *mover* puede usarse para una silla o una mesa, debido a que ambas son instancias de la clase **Mobiliario.** Todas las operaciones válidas (por ejemplo, *comprar, vender, pesar)* de la clase **Mobiliario** están «conectadas» a la definición del objeto como se muestra en la Figura 20.3 y son heredadas por todas las instancias de esta clase.

Referencia cruzada

Se puede utilizar la notación de modelado de datos del Capitulo 12.

El objeto **silla** (y todos los objetos en general) encapsula datos (los valores de los atributos que definen la silla), operaciones (las acciones que se aplican para cambiar los atributos de la silla), otros objetos (pueden definirse objetos compuestos [EVB89]), constantes (para fijar valores) y otra información relacionada. El *encapsulamiento* significa que toda esta información se encuentra empaquetada bajo un nombre y puede reutilizarse como una especificación o componente de programa.

Cita:

La encapsulación evita que un programa sea tan interdependiente que el más mínimo cambio provoque efectos devastadores.
Jim Rumbaugh et al.

Ahora que hemos introducido algunos conceptos básicos, resultará más significativa una definición más formal de la «orientación a objetos». Coad y Yourdon [COA91] definen el término de la siguiente forma:

orientación a objetos =
= objetos + clasificación + herencia + comunicación

Ya hemos introducido tres de estos conceptos. Pospondremos el tratamiento sobre la comunicación para más adelante.

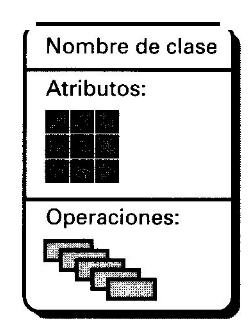
20.2.1. Clases y objetos

Los conceptos fundamentales que llevan a un diseño de alta calidad (Capítulo **13)** son igualmente aplicables a sistemas desarrollados usando métodos convencionales y orientados a objetos. Por esta razón, un modelo OO de software de computadora debe exhibir abstracciones de datos y procedimientos que conducen a una modularidad eficaz. Una clase es un concepto OO que encapsula las abstracciones de datos y procedimientos que se requieren para describir el contenido y comportamiento de alguna entidad del mundo real. Taylor [TAY90] usa la notación que se muestra a la derecha de la Figura 20.4 para describir una clase (y objetos derivados de una clase).



Un objeto encapsula datos (atributos) y los métodos (operaciones, métodos o servicios) que manipulan esos datos.

Las abstracciones de datos (atributos) que describen la clase están encerradas por una «muralla» de abstracciones procedimentales (llamadas operaciones, métodos o servicios) capaces de manipular los datos de alguna manera. La Única forma de alcanzar los atributos (y operar sobre ellos) es ir a través de alguno de los métodos que forman la muralla. Por lo tanto, la clase encapsula datos (dentro de la muralla) y el proceso que manipula los datos (los métodos que componen la muralla). Esto posibilita la ocultación de información y reduce el impacto de efectos colaterales asociados a cambios. Como estos métodos tienden a manipular un número limitado de atributos, esto es una alta cohesión, y como la comunicación ocurre sólo a través de los métodos que encierra la «muralla», la clase tiende a un desacoplamiento con otros elementos del sistema. Todas estas características del diseño (Capítulo 13) conducen a software de alta calidad.



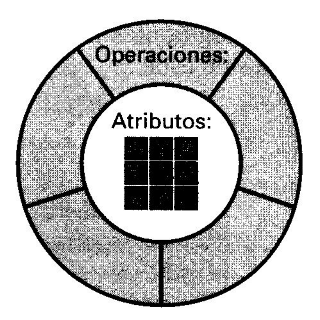


FIGURA 20.4. Representación alternativa de una clase orientada a objetos.

Puesto de otra manera, una clase es una *descripción generalizada* (por ejemplo, una plantilla, un patrón o un prototipo) que describe una colección de objetos similares. Por definición, todos los objetos que existen dentro de una clase heredan sus atributos y las operaciones disponibles para la manipulación de los atributos. Una *superclase* es una colección de clases y una *subclase* es una instancia de una clase.



Una de los primeros cosas a tener en cuenta a la hora de construir un sistema OO es cómo clasificar los objetos a manipular en dicho sistema.

Estas definiciones implican la existencia de una jerarquía de clases en la cual los atributos y operaciones de la superclase son heredados por subclases que pueden añadir, cada una de ellas, atributos «privados» y métodos. Una jerarquía de clases para la clase **Mobiliario** se ilustra en la Figura 20.5.

20.2.2. Atributos

Ya hemos visto que los atributos están asociados a clases y objetos, y que describen la clase o el objeto de alguna manera. Un estudio de los atributos es presentado por de Champeaux y sus colegas [CHA93]:

> Las entidades de la vida real están a menudo descritas con palabras que indican característicasestables. La mayoría de los objetos físicos tienen características tales como forma, peso, color y tipo de material. Las personas tienen características como fecha de nacimiento, padres, nombre y color de los ojos. Una característicapuede verse como una relación binaria entre una clase y cierto dominio.

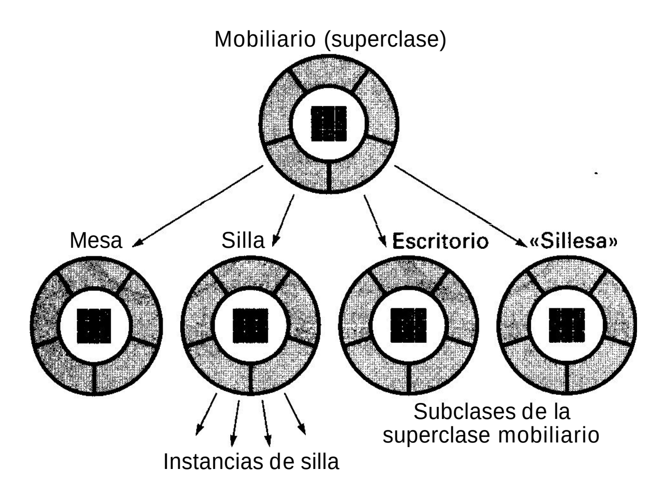


FIGURA 20.5. Una jerarquía de clases.

La «relación» binaria anteriormente señalada implica que un atributo puede tomar un valor definido por un dominio enumerado. En la mayoría de los casos, un dominio es simplemente un conjunto de valores específicos. Por ejemplo, suponga que una clase **Coche** tiene un atributo **color.** El dominio de valores de **color** es **(blanco, negro, plata, gris, azul, rojo, amarillo, verde).** En situaciones más complejas, el dominio puede ser un conjunto de clases. Continuando el ejemplo, la clase **Coche** también tiene un atributo **motor** que abarca los siguientes valores de dominio: **{ 16 válvulas opción económica, 16 válvulas opción deportiva, 24 válvulas opción deportiva, 32 válvulas opción de lujo).** Cada una de las opciones indicadas tiene un conjunto de atributos específicos.



los valores asignados a los atributos de un objeto hacen a ese objeto ser único.

Las características (valores del dominio) pueden aumentarse asignando un valor por defecto (característica) a un atributo. Por ejemplo, el atributo motor destacado antes, tiene el valor por defecto **16 válvulas opción deportiva.** Esto puede ser también Útil para asociar una probabilidad con una característica particular a través de pares {valor, probabilidad}. Considere el atributo **color** para **Coche.** En algunas aplicaciones (por ejemplo, planificación de la producción) puede ser necesario asignar una probabilidad a cada uno de los colores (blanco y negro son más probables como colores de coches).

20.2.3. Operaciones, métodos y servicios

Un objeto encapsula datos (representados como una colección de atributos) y los algoritmos que procesan estos datos. Estos algoritmos son llamados operaciones, métodos o servicios[1] y pueden ser vistos como módulos en un sentido convencional.



Siempre que un objeto es estimulado por un mensaje, inicia algún comportamiento ejecutando uno operación.

Cada una de las operaciones encapsuladas por un objeto proporciona una representación de uno de los comportamientos del objeto. Por ejemplo, la operación *DeterminarColor* para el objeto **Coche** extraerá el color almacenado en el atributo **color.** La consecuencia de la existencia de esta operación es que la clase **Coche** ha sido diseñada para recibir un estímulo [JAC92] (llamaremos al estímulo ***mensaje)*** que requiere el color de una instancia particular de una clase. Cada vez que un objeto recibe un estímulo, éste inicia un cierto comportamiento, que puede ser tan simple como determinar el color del coche o tan complejo como la iniciación de una cadena de estímulos que se pasan entre una variedad de objetos diferentes. En este último caso, considere un ejemplo en el cual el estímulo inicial recibido por el objeto n.º **1** da lugar a una generación de otros dos estímulos que se envían al objeto n.º 2 y al objeto n.º **3.** Las operaciones encapsuladas por el segundo y tercer objetos actúan sobre el estímulo devolviendo información necesaria para el primer objeto. El objeto n.º 1 usa entonces la información devuelta para satisfacer el comportamiento demandado por el estímulo inicial.

20.2.4. Mensajes

Los mensajes son el medio a través del cual interactúan los objetos. Usando la terminología presentada en la sección precedente, un mensaje estimula la ocurrencia de cierto comportamiento en el objeto receptor. El comportamiento se realiza cuando se ejecuta una operación.

[1] Usaremos aquí el término operaciones, pero métodos y servicios son igualmente populares.

En la Figura 20.6. se ilustra esquemáticamente la interacción entre objetos. Una operación dentro de un objeto emisor genera un mensaje de la forma:

destino.operación (parámetros)

donde *destino* define el *objeto receptor* el cual es estimulado por el mensaje, *operación* se refiere al método que recibe el mensaje y *parúmetros* proporciona información requerida para el éxito de la operación.

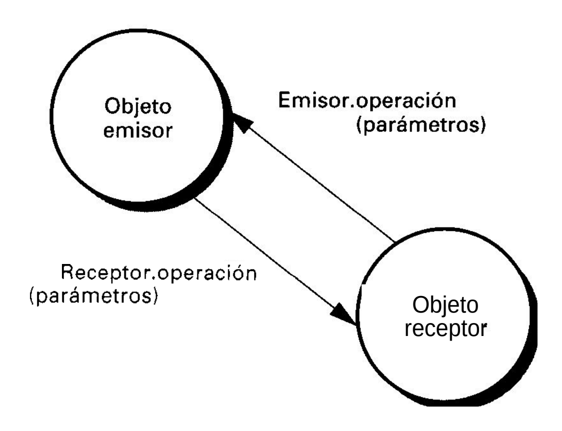


FIGURA 20.6. Paso de mensaje entre objetos.

Como un ejemplo de paso de mensajes dentro de un sistema **OO,** considere los objetos que se muestran en la Figura 20.7. Los cuatro objetos. **A, B, C** y D se comunican unos con otros a través del paso de mensajes. Por ejemplo, si el objeto **B** requiere el proceso asociado con la operación *op10* del objeto D, el primero enviaría a D un mensaje de la forma:

D.op 10(datos)



Los mensajes y los métodos son dos caras de la misma moneda. Los métodos son los procedimientos invocados cuando un objeto recibe un mensaje.
Greg Voss

Como parte de la ejecución de *op10*, el objeto **D** puede enviar un mensaje al objeto **C** de la forma:

C.op8(datos)

C encuentra op8, la ejecuta, y entonces envía un valor de retorno apropiado a **D**. La operación *op10* completa su ejecución y envía un valor de retorno a **B**.

Cox [COX86] describe el intercambio entre objetos de la siguiente manera:

> **Se solicita de un objeto que ejecute una de sus operaciones enviándole un mensaje que le informa acerca de lo que debe hacer. El [objeto] receptor responde al mensaje eligiendo primero la operación que implementa el nombre del mensaje, ejecutando dicha operación y después devolviendo el control al objeto que origina la llamada.**

El paso de mensajes mantiene comunicado un sistema orientado a objetos. Los mensajes proporcionan una visión interna del comportamiento de objetos individuales, y del sistema OO como un todo.

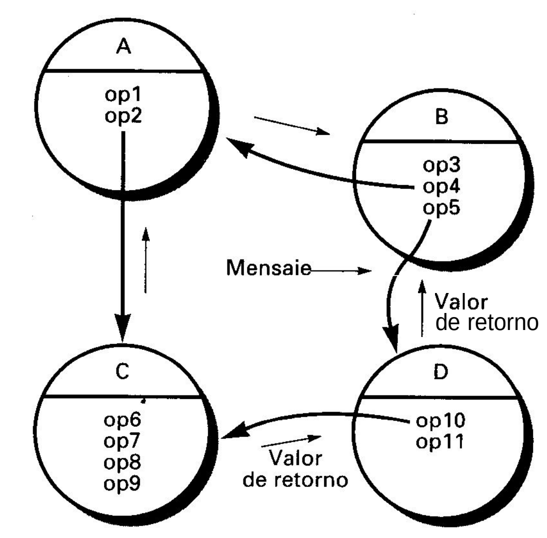
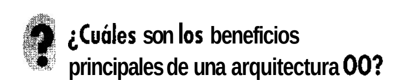


FIGURA 20.7. Paso de mensajes entre objetos.

20.2.5. Encapsulamiento, herencia y polimorfismo

Aunque la estructura y terminología introducida entre las Secciones 20.2.1 y 20.2.4 diferencian los sistemas OO a partir de sus componentes convencionales, hay tres características de los sistemas orientados a objetos que los hacen Únicos. Como ya hemos observado, las clases y los objetos derivados de ella encapsulan los datos y las operaciones que trabajan sobre estos en un único paquete. Esto proporciona un número importante de beneficios:



- Los detalles de implementación interna de datos y procedimientos están ocultos al mundo exterior (ocultación de la información). Esto reduce la propagación de efectos colaterales cuando ocurren cambios.
- Las estructuras de datos y las operaciones que las manipulan están mezcladas en una entidad sencilla: la clase. Esto facilita la reutilización de componentes.
- Las interfaces entre objetos encapsulados están simplificadas. Un objeto que envía un mensaje no tiene que preocuparse de los detalles de las estructuras de datos internas en el objeto receptor, lo que simplifica la interacción y hace que el acoplamiento del sistema tienda a reducirse.

La herencia es una de las diferencias clave entre sistemas convencionales y sistemas OO. Una subclase **Y** hereda todos los atributos y operaciones asociadas con su superclase **X.** Esto significa que todas las estructuras de datos y algoritmos originalmente diseñados e

implementados para **X** están inmediatamente disponibles para **Y** (no es necesario más trabajo extra). La reutilización se realiza directamente.

Cualquier cambio en los datos u operaciones contenidas dentro de una superclase es heredado inmediatamente por todas las subclases que se derivan de la superclase[2]. Debido a esto, la jerarquía de clases se convierte en un mecanismo a través del cual los cambios (a altos niveles) pueden propagarse inmediatamente a través de todo el sistema.

Es importante destacar que en cada nivel de la jerarquía de clases, pueden añadirse nuevos atributos y operaciones a aquellos que han sido heredados de niveles superiores de la jerarquía. De hecho, cada vez que se debe crear una nueva clase, el ingeniero del software tiene varias opciones:

- La clase puede diseñarse y construirse de la nada. Esto es, no se usa la herencia.
- La jerarquía de clases puede ser rastreada para determinar si una clase ascendiente contiene la mayoría de los atributos y operaciones requeridas. La nueva clase hereda de su clase ascendiente, y pueden añadirse otros elementos si hacen falta.
- La jerarquía de clases puede reestructurarse de tal manera que los atributos y operaciones requeridos puedan ser heredados por la nueva clase.
- Las características de una clase existente pueden sobrescribirse y se pueden implementar versiones privadas de atributos u operaciones para la nueva clase.

Cita:
Mientras que un objeto es una entidad que existe en el tiempo y en el espacio, una clase representa sólo una abstracción, «la esencia» del objeto, si se puede decir así.
Grady Booch

Para ilustrar cómo la reestructuración de la jerarquía de clases puede conducir a la clase deseada, considere el ejemplo mostrado en la Figura 20.8. La jerarquía de clases ilustradas en la Figura 20.8a nos permite derivar las clases **X3** y **X4** con las características 1, 2, 3, 4, 5 y 6, y 1, 2, 3, 4, 5 y 7, respectivamente[3]. Ahora, suponga que se desea crear una nueva clase solamente con las características 1, 2, 3, 4 y **8.** Para derivar esta clase, llamada **X2b** en el ejemplo, la jerarquía debe reestructurarse como se muestra en la Figura 20.8b. Es importante darse cuenta de que la reestructuración de la jerarquía puede ser difícil y, por esta razón, se usa a veces la *anulación.*

En esencia, la anulación ocurre cuando los atributos y operaciones se heredan de manera normal, pero después son modificados según las necesidades específicas de la nueva clase. Como señala Jacobson, cuando se emplea la anulación, «la herencia no es transitiva» [JAC92].

En algunos casos, es tentador heredar algunos atributos y operaciones de una clase y otros de otra clase. Esta acción se llama *herencia múltiple* y es controvertida. En general, la herencia múltiple complica la jerarquía de clases y crea problemas potenciales en el control de la configuración (Capitulo 9). Como las secuencias de herencia múltiple son más difíciles de seguir, los cambios en la definición de una clase que reside en la parte superior de la jerarquía pueden tener impactos no deseados originalmente en las clases definidas en zonas inferiores de la arquitectura.

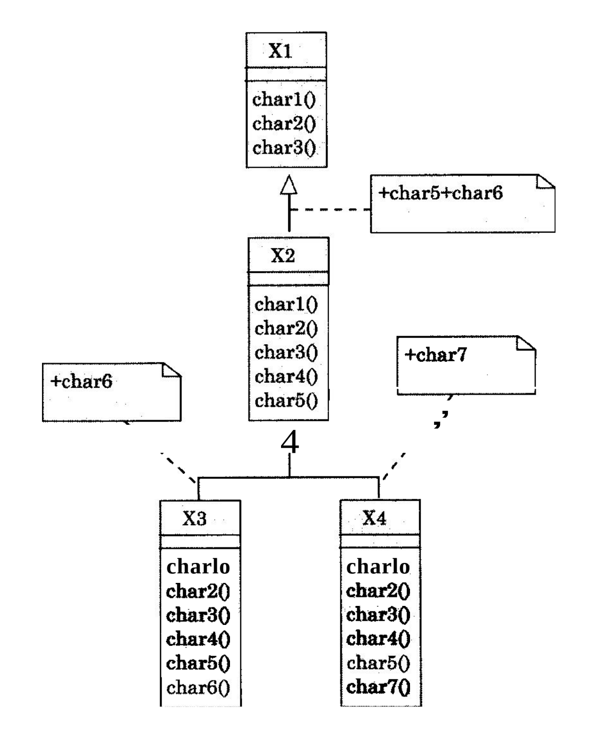


FIGURA 20.8a. Jerarquía de clases original.

Cita:
El nombre [polimorfismo] puede ser extraño, pero el mecanismo es puro elegancia.
David Taylor

El *polimorfismo* es una característica que reduce en gran medida el esfuerzo necesario para extender un sistema OO. Para entender el polimorfismo, considere una aplicación convencional que debe dibujar cuatro tipos diferentes de gráficos: gráficos de líneas, gráficos de tarta, histogramas y diagramas de Kiviat. Idealmente, una vez que se han recogido los datos necesarios para un tipo particular de gráfico, el gráfico debe dibujarse él mismo. Para realizar esto en una aplicación convencional (y man-

[2] A veces se emplean los términos descendiente y antecesor [JAC92] en lugar de subclase y superclase.

[3] En este ejemplo llamaremos característica tanto a los atributos como a las operaciones.

tener la cohesión entre módulos), sería necesario desarrollar módulos de dibujo para cada tipo de gráfico. Según esto, dentro del diseño para cada tipo de gráfico, se deberá añadir una lógica de control semejante a la siguiente:

```
case of tipo-grafico:
   if tipo-grafico = grafico_linea then
   DibujarLinea(datos);
   if tipo-grafico = grafico_tarta then
   DibujarTarta(datos);
   if tipo-grafico = grafico_histograma
   then DibujarHisto(datos);
   if tipo-grafico = grafico_kiviat then
   DibujarKiviat(datos);
end case;
```

Aunque este diseño es razonablemente evidente, añadir nuevos tipos de gráficos puede ser complicado, pues hay que crear un nuevo módulo de dibujo para cada tipo de gráfico y actualizar la lógica de control para cada gráfico.

Para resolver este problema, cada uno de los gráficos mencionados anteriormente se convierte en una subclase de una clase general llamada **Gráfico.** Usando un concepto llamado *sobrecarga* [TAY90], cada subclase define una operación llamada *dibujar.* Un objeto puede enviar un mensaje *dibujar* a cualquiera de los objetos instanciados de cualquiera de las subclases. El objeto que recibe el mensaje invocará su propia operación *dibujar* para *crear* el gráfico apropiado. Por esto, el diseño mostrado anteriormente se reduce a:

```
tipo-grafico dibujar
```

Cuando hay que añadir un nuevo tipo de gráfico al sistema, se crea una subclase con su propia operación

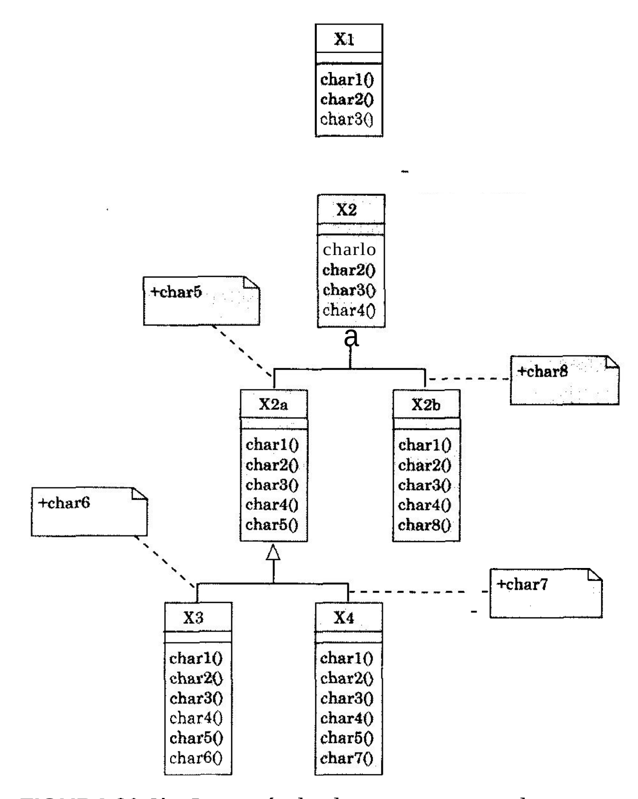
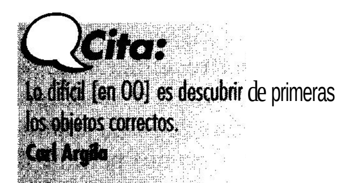


FIGURA 20.8b. Jerarquía de clases reestructurada.

20.3 IDENTIFICACIÓN DE LOS ELEMENTOS DE UN MODELO DE OBJETOS

Los elementos de un modelo de objetos -clases y objetos, atributos, operaciones y mensajes— fueron definidos y examinados en la sección precedente. Pero, ¿cómo podemos hacer para identificar estos elementos en un problema real? Las secciones que siguen presentan una serie de directrices informales que nos ayudarán en la identificación de los elementos de un modelo de objetos.



20.3.1. Identificación de clases y objetos

Si usted observa a su alrededor en una habitación, existen un conjunto de objetos físicos que pueden ser fácilmente identificados, clasificados y definidos (en términos de atributos y operaciones). Pero cuando usted «observa» el espacio de un problema en una aplicación de software, los objetos pueden ser más difíciles de identificar.

Podemos empezar a identificar objetos[4] examinando el planteamiento del problema o (usando la terminología del Capítulo 12) realizando un «análisis sintáctico gramatical» en la narrativa del sistema que se va a construir. Los objetos se determinan subrayando cada nombre o cláusula nominal e introduciéndola en una tabla simple. Los sinónimos deben destacarse. Si se requiere del objeto que implemente una solución, entonces

[4] En realidad, el Análisis OO intenta identificar las clases a partir de las cuales se instancian los objetos. Por tanto, cuando aislamos objetos potenciales, también identificamos miembros de clases potenciales.

éste formará parte del espacio de solución; en caso de que el objeto se necesite solamente para describir una solución, éste forma parte del espacio del problema. Pero, ¿qué debemos buscar una vez que se han aislado todos los nombres?

¿Cómo identificar objetos cuando estudio cómo resolver un problema?

Los objetos se manifiestan de alguna de las formas mostradas en la Figura 20.9. y pueden ser:

- *entidades externas* (por ejemplo: otros sistemas, dispositivos, personas) que producen o consumen información a usar por un sistemas computacional;
- *cosas* (por ejemplo: informes, presentaciones, cartas, señales) que son parte del dominio de información del problema;
- *ocurrencias* o *sucesos*[5] (por ejemplo: una transferencia de propiedad o la terminación de una serie de movimientos en un robot) que ocurren dentro del contexto de una operación del sistema;
- *papeles* o *roles* (por ejemplo: director, ingeniero, vendedor) desempeñados por personas que interactúan con el sistema;
- *unidades organizacionales* (por ejemplo: división, grupo, equipo) que son relevantes en una aplicación;
- *lugares* (por ejemplo: planta de producción o muelle de carga) que establecen el contexto del problema y la función general del sistema;
- *estructuras* (por ejemplo: sensores, vehículos de cuatro ruedas o computadoras) que definen una clase de objetos o, en casos extremos, clases relacionadas de objetos.

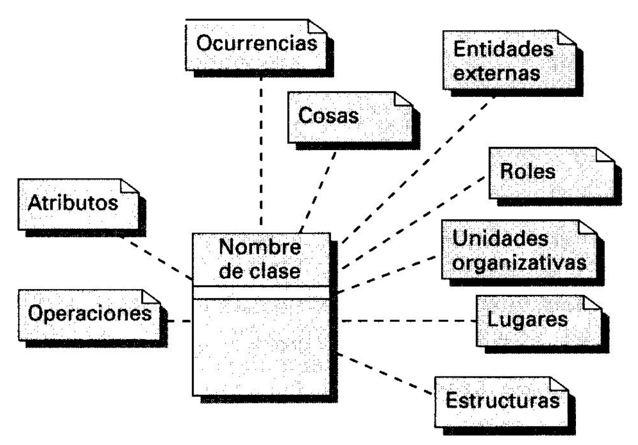


FIGURA 20.9. Cómo se manifiestan **los objetos.**

La clasificación mostrada anteriormente es una de las muchas que se han propuesto en la literatura.

También es importante destacar.qué no son los objetos. En general, un objeto nunca debe tener un «nombre procedimental imperativo» [CAS89]. **Por** ejemplo, si los desarrolladores de un software para un sistema gráfico médico definieron un objeto con el nombre **inversión de imagen,** estarían cometiendo un sutil error. La imagen obtenida por el software pudiera ser, en efecto, un objeto (es un elemento que forma parte del dominio de información), pero la inversión de la imagen es una operación que se aplica a dicho objeto. ***Inversión*** debe definirse como una operación del objeto **imagen,** pero no como objeto separado que signifique &versión de imagen». Como establece Cashman [CAS89], «...el objetivo de la orientación a objetos es encapsular, pero manteniendo separados los datos y las operaciones sobre estos datos».

Para ilustrar cómo pueden definirse los objetos durante las primeras etapas del análisis, volvamos al ejemplo del sistema de seguridad *HogarSeguro*. En el Capitulo 12, realizamos un «análisis sintáctico gramatical» sobre la narrativa de procesamiento para el sistema *HogarSeguro.* La narrativa de procesamiento se reproduce a continuación:

> El software *HogarSeguro* le permite al propietario de la casa configurar el sistema de seguridad una vez que este se instala, controla todos **los** sensores conectados al sistema de seguridad, e interactúa con el propietario a través de un teclado numérico y teclas de función contenidas en el panel de control de *HogarSeguro* mostrado en la Figura **11.2**
>
> Durante la instalación, el panel de control de *HogarSeguro* se usa para «programar» y configurar el sistema. **A** cada sensor se le asigna un número y tipo, se programa una contraseña maestra para activar y desactivar el sistema, y se **intro**ducen números de teléfono a marcar cuando un sensor detecte un suceso.
>
> Cuando se reconoce un suceso de sensor, el software invoca una alarma audible asociada al sistema. Después de un tiempo de espera especificado **por** el propietario durante las actividades de configuración del sistema, el software marca un número de teléfono de un servicio de control, proporciona información acerca de la localización, e informa de la naturaleza del suceso detectado. **El** número será remarcado cada 20 segundos hasta obtener una conexión telefónica.
>
> Toda la interacción con *HogarSeguro* está gestionada por un subsistema de interacción con el usuario que toma la entrada a partir del teclado numérico y teclas de función, y muestra mensajes y el estado del sistema en la pantalla LCD. La interacción con el teclado toma la siguiente forma **...**



Hay que utilizar un **analizador sintáctico** gramatical para detectar objetos potenciales (nombres) y operaciones (verbos).

[5] En este contexto, el termino suceso denota cualquier ocurrencia. No necesariamente implica control, en el sentido del Capítulo 12.

Extrayendo los nombres podemos proponer varios objetos potenciales:

Clase / Objeto potencial	Clasificación general
propietario	rol o entidad externa
sensor	entidad externa
panel de control	entidad externa
instalación	ocurrencia
sistema (alias sistema de seguridad)	cosa
número, tipo	no son objetos, sino atributos de sensor
contraseña maestra	cosa
número de teléfono	cosa
suceso de sensor	ocurrencia
alarma audible	entidad externa
servicio de control	unidad organizacional o entidad

La lista anterior se continuará hasta que se hayan considerado todos los nombres de la descripción del proceso. Observe que hemos llamado a cada entrada en la lista un objeto potencial. Debemos considerar cada uno de ellos antes de tomar una decisión final.

Cómo saber si un objeto potencial es un buen candidato para utilizarlo en un sistema OO.

Coad y Yourdon [COA91] sugieren seis características de selección a usar cada vez que un analista considera si incluye o no un objeto potencial en el modelo de análisis:

1. *Información retenida:* el objeto potencial será de utilidad durante el análisis solamente si la información acerca de él debe recordarse para que el sistema funcione.
2. *Servicios necesarios:* el objeto potencial debe poseer un conjunto de operaciones identificables que pueden cambiar de alguna manera el valor de **sus** atributos.
3. *Atributos múltiples:* durante el análisis de requisitos, se debe centrar la atención en la información principal (un objeto con un solo atributo puede, en efecto, ser Útil durante el diseño, pero seguramente será mejor presentado como un atributo de otro objeto durante la actividad de análisis);
4. *Atributos comunes:* puede definirse un conjunto de atributos para el objeto potencial, los cuales son aplicables a todas las ocurrencias del objeto.
5. *Operaciones comunes:* puede definirse un conjunto de operaciones para el objeto potencial, las cuales son aplicables a todas las ocurrencias del objeto;
6. *Requisitos esenciales:* entidades externas que aparecen en el espacio del problema y producen o consumen información esencial para la producción de cualquier solución para el sistema, serán casi siempre definidas como objetos en el modelo de requisitos.



CLAVE

Un objeto potencial debe satisfacer la mayoría de estas características para utilizarlo en el modelo de análisis.

Para ser considerado un objeto válido a incluir en el modelo de requisitos, un objeto potencial debe satisfacer todas (o casi todas) las características anteriores. La decisión de incluir objetos potenciales en el modelo de análisis es algo subjetivo, y una evaluación posterior puede llegar a descartar un objeto o reincluirlo. Sin embargo, el primer paso del AOO debe ser la definición de objetos, y la consiguiente toma de decisiones (incluso subjetivas). Teniendo esto en cuenta, aplicamos las características selectivas a la lista de objetos potenciales de *HogarSeguro* (véase tabla a continuación).

Clase / Objeto potencial	Características aplicables
propietario	Rechazado (1, 2 fallan aunque 6 es aplicable)
sensor	Aceptado (se aplican todas)
panel de control	Aceptado (se aplican todas)
instalación	Rechazado
sistema (alias sistema de seguridad)	Aceptado (se aplican todas)
número, tipo	Rechazado (falla 3)
contraseña maestra	Rechazado (falla 3)
número de teléfono	Rechazado (falla 3)
suceso de sensor	Aceptado (se aplican todas)
alarma audible	Aceptado (se aplican 2, 3, 4, 5 Y 6)
servicio de control	Rechazado (fallan 1 y 2 aunque 6 se aplica)

Debe tenerse en cuenta que: (1) la lista anterior no incluye todo, hay que añadir objetos adicionales para completar el modelo; (2) algunos de los objetos potenciales rechazados serán atributos de los objetos aceptados (por ejemplo, número y tipo son atributos de **sensor,** y contraseña maestra y número de teléfono pueden convertirse en atributos de **sistema);** y **(3)** diferentes descripciones del problema pueden provocar la toma de diferentes decisiones de «aceptación o rechazo» (por ejemplo, si cada propietario tiene su propia contraseña o fue identificado por impresiones de voz, el objeto **Propietario** cumpliría las características **1** y 2 y habría sido aceptado).

20.3.2. Especificación de atributos

Los atributos describen un objeto que ha sido seleccionado para ser incluido en el modelo de análisis. En esencia, son los atributos los que definen al objeto, los que clarifican lo que se representa el objeto en el contexto del espacio del problema. Por ejemplo, si tratáramos de construir un sistema de estadísticas para jugadores profesionales de béisbol, los atributos del objeto **Jugador** serían muy diferentes de los atributos del mismo objeto cuando se use dentro del contexto de un sistema de pensiones para jugadores profesionales. En el primero, atributos tales como nombre, posición; promedio de bateo, porcentaje de estancia en el campo de juego, años jugados y partidos jugados pueden ser relevantes. En el segundo caso, algunos de estos atributos serían relevantes pero otros serían reemplazados (o potenciados) por atributos como salario medio, crédito total, opciones elegidas para el plan de pensión, dirección postal, etc.

Para desarrollar un conjunto significativo de atributos para un objeto, el analista puede estudiar de nuevo la narrativa del proceso (o descripción del ámbito del alcance) para el problema y seleccionar aquellos elementos que razonablemente «pertenecen» al objeto. Además, para cada objeto debe responderse el siguiente interrogante: «¿Qué elementos (compuestos y/o simples) definen completamente al objeto en el contexto del problema actual?»



Los atributos se escogen examinando el problema, buscando cosas que definan completamente los objetos y que los hacen únicos.

Para ilustrar esto, consideremos el objeto **Sistema** definido para *HogarSeguro*. Anteriormente ya indicamos que el propietario puede configurar el sistema de seguridad para reflejar la información acerca de los sensores, sobre la respuesta de la alarma, sobre la activación/desactivación, sobre identificación, etc. Usando la notación de la descripción del contenido definida para el diccionario de datos y presentada en el Capítulo 12, podríamos representar estos elementos de datos compuestos de la siguiente manera:

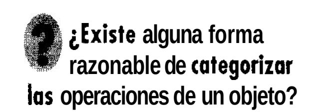
```
información del censor = tipo de sensor
+ número de sensor + umbral de alarma
información de respuesta de la alarma =
tiempo de retardo + número de teléfono
+ tipo de alarma
información de activación/desactivación
= contraseña maestra + cantidad de
intentos permitidos + contraseña tempo-
ral
información de identificación = ID del
sistema + verificación de número de telé-
Eono + estado del sistema
```

Cada uno de los elementos de datos a la derecha del signo de igualdad puede volver a definirse en un nivel elemental, pero para nuestros propósitos, comprenden una lista razonable de atributos para el objeto sistema (porción sombreada de la Fig. 20.10).

20.3.3. Definición de operaciones

Las operaciones definen el comportamiento de un objeto y cambian, de alguna manera, los atributos de dicho objeto. **Más** concretamente, una operación cambia valores de uno o más atributos contenidos en el objeto. Por lo tanto, una operación debe tener «conocimiento» de la naturaleza de los atributos de los objetos y deben ser implementadas de manera tal que le permita manipular las estructuras de datos derivadas de dichos atributos.

Aunque existen muchos tipos diferentes de operaciones, éstas pueden clasificarse en tres grandes categorías: (1) operaciones que manipulan, de alguna manera, datos (p.e.: añadiendo, eliminando, reformateando, seleccionando); (2) operaciones que realizan algún cálculo; y (3) operaciones que monitorizan un objeto frente a la ocurrencia de un suceso de control.



¿Existe alguna forma razonable de categorizar las operaciones de un objeto?

En una primera iteración para obtener un conjunto de operaciones para los objetos del modelo de análisis, el analista puede estudiar de nuevo la narrativa del proceso (o descripción del ámbito) del problema y seleccionar aquellas operaciones que razonablemente pertenecen al objeto. Para realizar esto, se estudia de nuevo el análisis gramatical y se aíslan los verbos. Algunos de estos verbos serán operaciones legítimas y pueden conectarse fácilmente a un objeto específico. Por ejemplo, de la narrativa de proceso de *Hogarseguro*, presentada anteriormente en este capítulo, vemos que cal sensor se le asigna un número y un tipo» o que «se programa una contraseña maestra para activar y desactivar el sistema». Estas dos frases indican varias cosas:

- que una operación de *asignación* es relevante para el objeto **Sensor;**
- que una operación de *programar* se le aplicará al objeto **Sistema;**

- que *activar* y *desactivar* son operaciones aplicables al **Sistema** (o sea que el **estado del sistema** puede en última instancia definirse usando notación del diccionario de datos) como

  estado del sistema = [activado | desactivado]

Tras una investigación más detallada, es probable que haya que dividir la operación *programar* en varias suboperaciones más específicas requeridas para configurar el sistema. Por ejemplo, *programar* implica especificar números de teléfonos, configurar características del sistema (por ejemplo, crear la tabla de sensores, introducir las características de las alarmas), e introducir la(s) contraseña(s), pero por ahora, especificaremos *programar* como una operación simple.

```
Sistema
-----------------------------------
ID del sistema
Número de teléfono de verificación
Estado del sistema
Tabla de sensores
Tipo de sensor
Número de sensor
Umbral de alarma
Contraseña maestra
Contraseña temporal
Número de intentos
-----------------------------------
Programar ( )
Mostrar ( )
Reiniciar ( )
Consultar ( )
Modificar ( )
Llamar ( )
```

FIGURA 20.10. El objeto **sistema con operaciones asociadas.**

20.3.4. Fin de la definición del objeto

La definición de operaciones es el Último paso para completar la especificación del objeto. En la Sección 20.3.3. las operaciones se eligieron a partir de un examen gramatical de la narrativa de proceso del sistema. Las operaciones adicionales pueden determinarse considerando la «historia de la vida» [COA91] de un objeto y los mensajes que se pasan entre objetos definidos por el sistema.

La historia de la vida genérica de un objeto puede definirse reconociendo que dicho objeto debe ser creado, modificado, manipulado o leído de manera diferente, y posiblemente borrado. Para el objeto **Sistema,** esto puede expandirse para reflejar actividades conocidas que ocurren durante su vida (en este caso, durante el tiempo que *HogarSeguro* se mantiene operativo). Algunas de las operaciones pueden determinarse a partir de comunicaciones semejantes entre objetos. Por ejemplo, el **Suceso sensor** enviará un mensaje a **Sistema** para *mostrar* en pantalla la localización y número del suceso; el **Panel de control** enviará un mensaje de *reinicialización* para actualizar el estado del **Sistema** (un atributo); la **Alarma audible** enviará un mensaje de *consulta*, el **Panel de control** enviará un mensaje de *modificación* para cambiar uno o más atributos sin reconfigurar el objeto sistema por completo; el **Suceso sensor** enviará un mensaje para *llamar* al número**(s)** de teléfono(s) contenido(s) en el objeto. Podrán considerarse otros mensajes para derivar operaciones correspondientes. La definición del objeto resultante se muestra en la Figura 20.10.

Se usaría un enfoque similar para cada uno de los objetos definidos para *HogarSeguro*. Después de haber definido los atributos y operaciones para todos los objetos especificados hasta este punto, se crearían los inicios del modelo AOO. En el Capítulo 21 se presenta un estudio más detallado del modelo de análisis creado como parte del AOO.

20.4 GESTIÓN DE PROYECTOS DE SOFTWARE ORIENTADO A OBJETOS

Como examinamos en la Parte Primera y Segunda de este libro, la moderna gestión de proyectos de software puede subdividirse en las siguientes actividades:

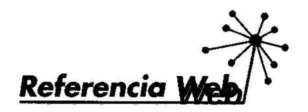
1. Establecimiento de un marco de proceso común para el proyecto.
2. Uso del marco y de métricas históricas para desarrollar estimaciones de esfuerzo y tiempo.
3. Especificación de productos de trabajo e hitos que permitirán medir el progreso.
4. Definición de puntos de comprobación para la gestión de riesgos, aseguramiento de la calidad y control.
5. Gestión de los cambios que ocurren invariablemente al progresar el proyecto.
6. Seguimiento, monitorización y control del progreso.



*las proyectas OO requieren mucha más gestión de lo **planificación** y seguimiento que los proyectas de **software** convencional. **No** suponga que la OO le releva de esto actividad.*

El director técnico que se enfrenta con un proyecto de software orientado a objetos aplica estas seis actividades. Pero debido a la naturaleza única del software orientado a objetos, cada una de estas actividades de gestión tiene un matiz levemente diferente y debe ser enfocado usando un modelo propio.

En las secciones que siguen, exploramos el área de gestión de proyectos orientados a objetos. Los principios de gestión fundamentales serán los mismos, pero para que un proyecto OO sea dirigido correctamente hay que adaptar la técnica.



Encontrará un tutorial muy completo sobre gestión de proyectos OO y un buen conjunto de enlaces en: **mini.net/cetus/oo_project_mngt.html.**

20.4.1. El marco de proceso común para OO

Un marco de proceso común (MPC) define un enfoque organizativo para el desarrollo y mantenimiento de software. El **MPC** identifica el paradigma de ingeniería del software aplicado para construir y mantener el software, así como las tareas, hitos y entregas requeridos. Establece el grado de rigor con el cual se enfocarán los diferentes tipos de proyectos. El MPC siempre es adaptable, de tal manera que siempre cumpla con las necesidades individuales del equipo del proyecto. Ésta es su característica más importante.

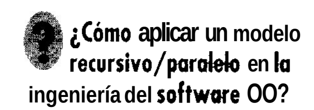
Referencia cruzada

El marco de procesa común **define** las actividades básicas de ingeniería **del** software. Se describe en el Capítulo 2.

Ed Berard [BER93] y Grady Booch [BOO91], entre otros, sugieren el uso de un «modelo recursivo/paralelo» para el desarrollo de software orientado a objetos. En esencia, el modelo recursivo/paralelo funciona de la siguiente manera:

- Realizar los análisis suficientes para aislar las clases del problema y las conexiones más importantes.
- Realizar un pequeño diseño para determinar si las clases y conexiones pueden ser implementadas de manera práctica.
- Extraer objetos reutilizables de una biblioteca para construir un prototipo previo.
- Conducir algunas pruebas para descubrir errores en el prototipo.
- Obtener realimentación del cliente sobre el prototipo.
- Modificar el modelo de análisis basándose en lo que se ha aprendido del prototipo, de la realización del diseño y de la realimentación obtenida del cliente.
- Refinar el diseño para acomodar sus cambios.
- Construir objetos especiales (no disponibles en la biblioteca).
- Ensamblar un nuevo prototipo usando objetos de la biblioteca y los objetos que se crearon nuevos.
- Realizar pruebas para descubrir errores en el prototipo.
- Obtener realimentación del cliente sobre el prototipo.

Este enfoque continúa hasta que el prototipo evoluciona hacia una aplicación en producción.



¿Cómo aplicar un modelo recursivo/paralelo en la ingeniería del software OO?

El modelo recursivo/paralelo es muy similar al modelo de proceso OO presentado anteriormente en este capítulo. El progreso se produce iterativamente. Lo que hace diferente al modelo recursivo/paralelo es el reconocimiento de que (1) el modelo de análisis y diseño para sistemas OO no puede realizarse a un nivel uniforme de abstracción, y (2) el análisis y diseño pueden aplicarse a componentes independientes del sistema de manera concurrente. Berard [BER93] describe el modelo de la siguiente manera:

- Descomponer sistemáticamente el problema en componentes altamente independientes.
- Aplicar de nuevo el proceso de descomposición a cada una de las componentes independientes para, a su vez, descomponerlas (la parte recursiva).
- Conducir este proceso de reaplicar la descomposición de forma concurrente sobre todos los componentes (la parte paralela).
- Continuar este proceso hasta cumplir los criterios de finalización.

Es importante observar que el proceso de descomposición mostrado anteriormente es discontinuo si el analista/diseñador reconoce que el componente o subcomponente requerido está disponible en una biblioteca de reutilización.

Para controlar el marco de proceso recursivo/paralelo, el director del proyecto tiene que reconocer que el progreso está planificado y medido de manera incremental. Esto es, las tareas y la planificación del proyecto están unidas a cada una de las «componentes altamente independientes», y el progreso se mide para cada una de estas componentes individualmente.



En muchos aspectos, lo arquitectura de un sistema OO hace que el comenzar o trabajar en paralelo sea más fácil. Sin embargo, también es cierto que codo toreo paralela se define de formo que pueda calcularse el progreso.

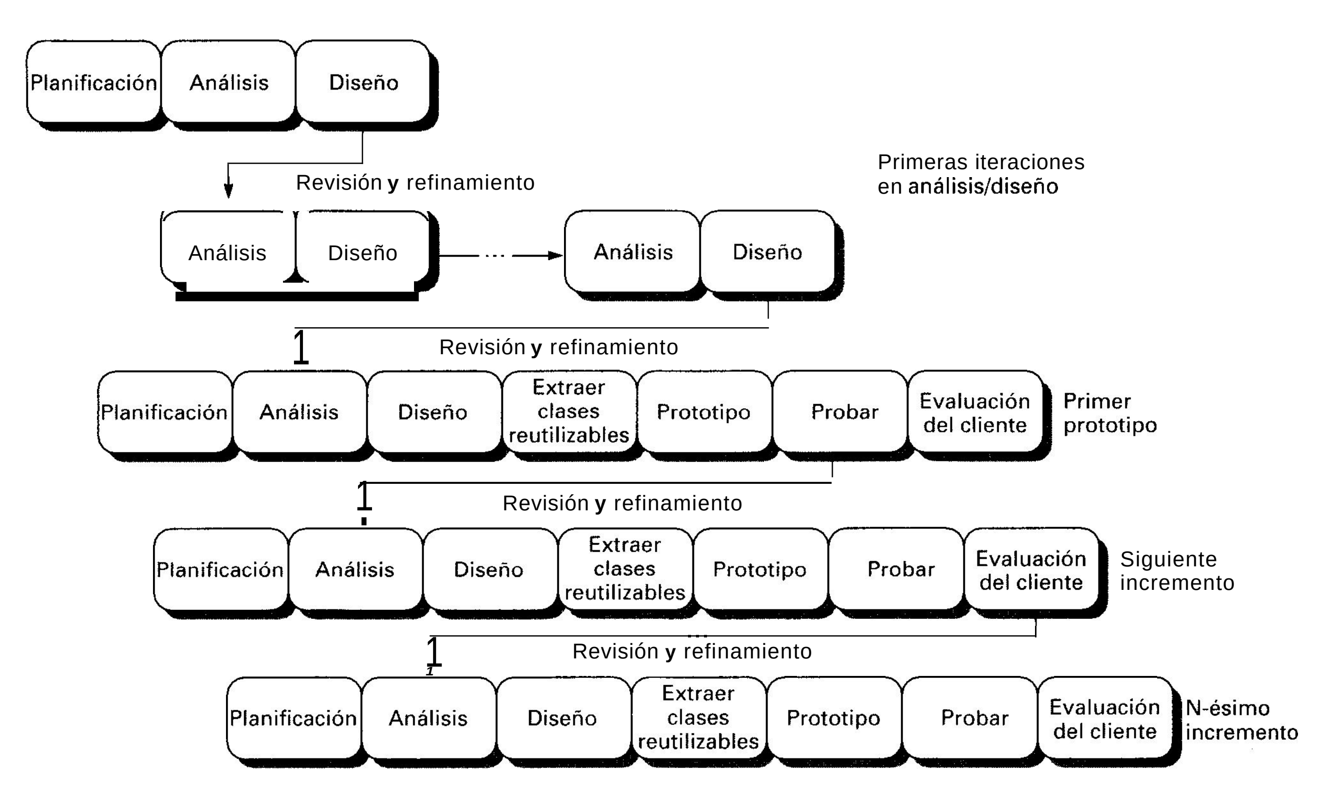


FIGURA 20.11. Secuencia típica de un proceso para un proyecto OO.

Cada iteración del proceso recursivo/paralelo requiere planificación, ingeniería (análisis, diseño, extracción de clases, prototipado y pruebas) y actividades de evaluación (Fig. 20.11). Durante la planificación, las actividades asociadas con cada una de las componentes independientes del programa son incluidas en la planificación. [Nota: Con cada iteración se ajusta la agenda para acomodar los cambios asociados con la iteración precedente]. Durante las primeras etapas del proceso de ingeniería, el análisis y el diseño se realizan iterativamente. La intención es identificar todos los elementos importantes del análisis OO y de los modelos de diseño. Al continuar el trabajo de ingeniería, se producen versiones incrementales del software. Durante la evaluación se realizan, para cada incremento, revisiones, evaluaciones del cliente y pruebas, las cuales producen una realimentación que afecta a la siguiente actividad de planificación y al subsiguiente incremento.

20.4.2. Métricas y estimación en proyectos orientados a objetos

Las técnicas de estimación en proyectos de software convencionales requieren estimaciones de líneas de código (LDC) o puntos de función (PF) como controlador principal de estimación. Las estimaciones realizadas a partir de LDC tienen poco sentido en proyectos OO debido a que el objetivo principal es la reutilización. Las estimaciones a partir de PF pueden usarse de manera efectiva, pues los elementos del dominio de información requeridos se pueden determinar a partir del planteamiento del problema. El análisis de PF puede aportar valores para estimaciones en proyectos **OO,** pero la medida de PF no provee una granularidad suficiente para ajustes de planificación y esfuerzo a realizar, los cuales se requieren cuando iteramos a través del paradigma recursivo/paralelo.

Lorenz y Kidd [LOR94] sugieren el siguiente conjunto de métricas para proyectos[6]:

Número de guiones de escenario. Un *guión de escenario* (como los casos de uso discutidos en el Capítulo 11) es una secuencia detallada de pasos que describen la interacción entre el usuario y la aplicación. Cada guión se organiza en tripletes de la forma:

{**iniciador,** *acción,* **participante]**

donde **iniciador** es el objeto que solicita algún servicio (que inicia un mensaje); *acción* es el resultado de la solicitud; y **participante** es el objeto servidor que cumple la petición (solicitud). El número de guiones de actuación está directamente relacionada al tamaño de la aplicación y al número de casos de prueba que se deben desarrollar para ejercitar el sistema una vez construido.

[6] Las técnicas de medida de sistemas OO se discuten con detalle en el capítulo 24.



*Estos métricos pueden utilizarse poro **complementar** los métricos FP. Proporcionan uno formo de «medir» un proyecto OO.*

Número de clases clave. Las *clases clave* son las «componentes altamente independientes» [LOR94] definidas inicialmente en el AOO. Debido a que estas clases son centrales al dominio del problema, el número de dichas clases es una indicación del esfuerzo necesario para desarrollar el software y de la cantidad potencial de reutilización a aplicar durante el desarrollo del sistema.

Número de clases de soporte. Las *clases de soporte* son necesarias para implementar el sistema, pero no están directamente relacionadas con el dominio del problema. Como ejemplos podemos tomar las clases de Interfaz Gráfica de Usuario, el acceso a bases de datos y su manipulación y las clases de comunicación. Además las clases de soporte se definen iterativamente a lo largo del proceso recursivo/paralelo.

El número de clases de soporte es un indicador del esfuerzo necesario requerido para desarrollar el software y de la cantidad potencial de reutilización a aplicar durante el desarrollo del sistema.

Referencia cruzada

En el Capítulo **24** se presentan detalladamente **las métricas** OO.

Número promedio de clases de soporte por clase clave. En general las clases clave son conocidas en las primeras etapas del proyecto. Las clases de soporte se definen a lo largo de éste. Si, dado un dominio de problema, se conociera la cantidad promedio de clases de soporte por clase clave, la estimación (basada en la cantidad total de clases) se simplificaría mucho.

Lorenz y Kidd sugieren que las aplicaciones con IGU poseen entre dos y tres veces más clases de soporte que clases clave. Las aplicaciones sin IGU poseen a lo sumo dos veces más de soporte que clave.

Número de subsistemas. Un *subsistema* es una agregación de clases que dan soporte a una función visible al usuario final del sistema. Una vez que se identifican los subsistemas, resulta más fácil realizar una planificación razonable en la cual el trabajo en los subsistemas está repartida entre los miembros del proyecto.

20.4.3. Un enfoque OO para estimaciones y planificación

La estimación en proyectos de software es más un arte que una ciencia. Sin embargo, esto en modo alguno excluye el uso de un enfoque sistemático. Para desarrollar estimaciones razonables es esencial el desarrollo de múltiples puntos de datos. Esto significa que las estimaciones deben derivarse usando diferentes técnicas. Las estimaciones respecto al esfuerzo y la duración usadas en el desarrollo de software convencional (Capítulo 5) son aplicables al mundo de la **OO,** pero la base de datos histórica para proyectos OO es relativamente pequeña en muchas organizaciones. Debido a esto, vale la pena sustituir la estimación de costes para software convencional por un enfoque diseñado explícitamente para software OO. Lorenz **y** Kidd [LOR94] sugieren el siguiente enfoque:

1. Desarrollo de estimaciones usando la descomposición de esfuerzos, análisis de PF y cualquier otro método aplicable a aplicaciones convencionales.
2. Desarrollar guiones de escenario y determinar una cuenta, usando AOO (Capítulo 21). Reconocer que el número de guiones de escenarios puede cambiar con el progreso del proyecto.

Referencia cruzada

En el Capítulo **5** se consideran diferentes técnicos de **estimación de proyectos** software

3. Determinar la cantidad de clases clave usando AOO.
4. Clasificar el tipo de interfaz para la aplicación y desarrollar un multiplicador para las clases de soporte:

Tipo de Interfaz	Multiplicador
Interfaz no gráfica (No IGU)	2,0
Interfaz de usuario basada en texto	2,25
Interfaz Gráfica de Usario (IGU)	2,5
Interfaz Gráfica de Usuario (IGU) compleja	3,0

   Multiplicar el número de clases clave (paso 3) por el multiplicador anterior para obtener una estimación del número de clases de soporte.
5. Multiplicar la cantidad total de clases (clave + soporte) por el número medio de unidades de trabajo por clase. Lorenz y Kidd sugieren entre 15 y 20 días-persona por clase.
6. Comprobar la estimación respecto a clases multiplicando el número promedio de unidades de trabajo por guión de acción.

La planificación de proyectos orientados a objetos es complicada por la naturaleza iterativa del marco de

trabajo del proceso. Lorenz y Kidd sugieren un conjunto de métricas que pueden ayudar durante esta planificación del proyecto:

Número de iteraciones principales. Recordando el modelo en espiral (Capítulo 2), una iteración principal corresponderá a un recorrido de 360 grados de la espiral. El modelo de proceso recursivo/paralelo engendrará un número de mini—espirales (iteraciones localizadas) que suceden durante el progreso de la iteración principal. Lorenz y Kidd sugieren que las iteraciones de entre 2.5 y 4 meses de duración son más fáciles de seguir y gestionar.

Número de contratos completos. Un contrato es «un grupo de responsabilidades públicas relacionadas que los subsistemas y las clases proporcionan a sus clientes» [LOR94]. Un contrato es un hito excelente, y debería asociarse al menos un contrato a cada iteración del proyecto. Un jefe de proyecto puede usar contratos completos como un buen indicador del progreso en un proyecto OO.

20.4.4. Seguimiento del progreso en un proyecto orientado a objetos

Aunque el modelo de proceso recursivo/paralelo es el mejor marco de trabajo para un proyecto OO, el paralelismo de tareas dificulta el seguimiento del proyecto. El jefe del proyecto puede tener dificultades estableciendo hitos significativos en un proyecto OO debido a que siempre hay un cierto número de cosas ocurriendo a la vez. En general, los siguientes hitos principales pueden considerarse «completados» al cumplirse los criterios mostrados:

Hito técnico: análisis OO terminado

- Todas las clases, y la jerarquía de clases, están definidas y revisadas.
- Se han definido y revisado los atributos de clase y las operaciones asociadas a una clase.
- Se han establecido y revisado las relaciones entre clases (Capítulo 21).
- Se ha creado y revisado un modelo de comportamiento (Capítulo 21).
- Se han marcado clases reutilizables.

Hito técnico: diseño OO terminado

- Se ha definido y revisado el conjunto de subsistemas (Capítulo 22).
- Las clases se han asignado a subsistemas y han sido revisadas.
- Se ha establecido y revisado la asignación de tareas.
- Se han identificado responsabilidades y colaboraciones (Capítulo 22).
- Se han diseñado y revisado los atributos y operaciones.
- Se ha creado y revisado el modelo de paso de mensajes.

Hito técnico: programación OO terminada

- Cada nueva clase ha sido implementada en código a partir del modelo de diseño.
- Las clases extraídas (de una biblioteca de reutilización) se han integrado.
- Se ha construido un prototipo o incremento.

Hito técnico: prueba OO

- Han sido revisadas la corrección y compleción del análisis 00 y del modelo de diseño.
- Se ha desarrollado y revisado una red de clases—responsabilidades–colaboraciones (Capítulo 23).
- Se han diseñado casos de prueba y ejecutado pruebas al nivel de clases para cada clase (Capítulo 23).
- Se han diseñado casos de prueba y completado pruebas de agrupamientos (Capítulo 23) y las clases se han integrado.
- Se han terminado las pruebas del sistema.

Recordando el modelo de proceso recursivo/paralelo examinado anteriormente en este capítulo, es importante destacar que cada uno de estos hitos puede ser revisado nuevamente al entregar diferentes incrementos al usuario.

RESUMEN

Las tecnologías de objetos reflejan una visión natural del mundo. Los objetos se clasifican en clases y las clases se organizan en jerarquías. Cada clase contiene un conjunto de atributos que la describen y un conjunto de operaciones que define su comportamiento. Los objetos modelan casi todos los aspectos identificables del dominio del problema: entidades externas, cosas, ocurrencias, roles, unidades organizacionales, lugares y estructuras pueden ser representados como objetos. Es importante destacar que los objetos (y las clases de las que se derivan) encapsulan datos y procesos. Las operaciones de procesamiento son parte del objeto y se inician al pasarle un mensaje al objeto. Una definición de clase, una vez definida, constituye la base para la reutilización en los niveles de modelado, diseño e implementación. Los objetos nuevos pueden ser instaciados a partir de una clase.

Tres conceptos importantes diferencian el enfoque OO de la ingeniería del software convencional. El encapsulamiento empaqueta datos y las operaciones que manejan esos datos. La herencia permite que **los** atributos y operaciones de una clase puedan ser heredados por todas las clases y objetos que se instancian de ella. El polimorfismo permite que una cantidad de operaciones diferentes posean el mismo nombre, reduciendo la cantidad de líneas de código necesarias para implementar un sistema y facilita los cambios en caso de que se produzcan.

Los productos y sistemas orientados a objetos se producen usando un modelo evolutivo, a veces llamado recursivo/paralelo. El software orientado a objetos evoluciona iterativamente y debe dirigirse teniendo en cuenta que el producto final se desarrollará a partir de una serie de incrementos.

REFERENCIAS

[BER93] Berard, E.V., ***Essays on Object—Oriented Software Engineering,*** Addison—Wesley, 1993.

[BOO86] Booch, G., «*Object-Oriented Development*», ***IEEE Trans. Software Engineering,*** Vol. SE-12, Febrero 1986, pp. 211 y **ss.**

[BOO91] Booch, G., ***Object-Oriented Design,*** Benjamin-Cummings, 1991.

[BUD96] Budd, T., ***An introduction to Object-Oriented Programming,*** 2.ª ed., Addison-Wesley, 1996.

[CAS89] Cashman, M., «Object-Oriented Domain Analysis», ***ACM Software Engineering Notes,*** vol. 14, n.º 6, Octubre 1989, pp 67.

[CHA93] Champeaux, D. de D. Lea, y P. Faure, ***Object-Oriented System Development,*** Addison-Wesley, 1993.

[COA91] Coad, P., y Yourdon, E., *Object-Oriented Analysis*, 2." ed., Prentice-Hall, 1991.

[COX86] Cox, B.J., ***Object-Oriented Programming,*** Addison-Wesley, 1986.

[EVB89] ***Object-Oriented Requirements Analysis*** (course notebook), EVB Software Engineering, Inc., Frederick, MD, 1989.

[JAC92] Jacobson, I., ***Object-Oriented Software Engineering,*** Addison-Wesley, 1992.

[LOR94] Lorenz, M., y Kidd, J., ***Object-Oriented Software Metrics,*** Prentice-Hall, 1994.

[STR88] Stroustrup, B., «What is Object-Oriented Programming?», ***IEEE Software,*** vol. **5**, n.º 3, Mayo 1988, pp. 10-20.

[TAY90] Taylor, D.A., ***Object-Oriented Technology: A Manager's Guide,*** Addison-Wesley, 1990.

PROBLEMAS Y PUNTOS A CONSIDERAR

20.1. La ingeniería del software orientada a objetos está reemplazando rápidamente a los enfoques de desarrollo de software tradicionales. Como todas las tecnologías, la orientación a objetos también tiene **sus** fallos. Utilizando Internet y otras fuentes de bibliografía más tradicionales, escriba un breve artículo que resuma lo que les críticos dicen sobre la OO y por qué creen que hay que tener cuidado al aplicar el paradigma de objetos.

20.2. En este capítulo no hemos considerado el caso en el que un nuevo objeto necesita un atributo u operación que no está contenido en la clase de la que hereda el resto de atributos y operaciones. ¿Cómo cree que se manejaría esta situación?

20.3. Detalle los objetos que participarían en un sistema de reservas de vuelos. ¿Cuáles serían sus atributos?

20.4. Utilizando sus propias palabras y algunos ejemplos defina los términos clase, encapsulamiento, herencia y polimorfismo.

20.5. Revise los objetos definidos en el sistema ***HogarSeguro.*** ¿Cree que hay otros objetos que deban ser definidos como principio del modelado?

20.6. Considere la típica interfaz gráfica de usuario (IGU). Defina un conjunto de clases y subclases para las entidades de la interfaz que aparecen generalmente en una IGU. Asegúrese de definir los atributos y operaciones apropiadas.

20.7. Detalle los objetos que aparecerían en un sistema de reserva de aulas de lectura en una universidad o colegio. ¿Cuáles serían sus atributos?

20.8. Le ha sido asignada la tarea de ingeniería de un nuevo programa de procesamiento de textos. Se ha identificado una clase llamada **documento.** Defina los atributos y operaciones relevantes de dicha clase.

20.9. Investigue dos lenguajes de programación OO diferentes y muestre la implementación de los mensajes en la sintaxis de cada uno de ellos. Ponga ejemplos.

20.10. Ponga un ejemplo concreto de reestructuración de jerarquía de clases tal y como aparece en la discusión de la Figura 20.8.

20.11. Ponga un ejemplo de herencia múltiple. Investigue algunos artículos sobre el tema y extraiga los pros y los contras.

20.12. Escriba un enunciado para un sistema que le proporcione su profesor. Utilice el analizador sintáctico gramatical para identificar clases candidatas, atributos y operaciones. Aplique el criterio de selección discutido en la Sección 20.3.1 para determinar qué clases deberían estar en el modelo de análisis.

20.13. Describa con sus propias palabras por qué el modelo recursivo/paralelo es apropiado en los sistemas OO.

20.14. Ponga tres o cuatro ejemplos de clases clave y de soporte que se describieron en la Sección 20.4.2.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

La explosión de interés por las tecnologías OO ha dado como resultado **la** publicación de literalmente cientos de libros durante los Últimos 15 años. El tratamiento abreviado de Taylor [TAY90] sigue siendo la mejor introducción al tema. Además, los libros de Ambler ***(The Object Primer: The application Developper's Guide to Object-Orientation,*** SIGS Books, 1998), Gossain y Graham ***(Object Modeling and Design Strategies,*** SIGS Books, 1998), Bahar ***(Object Technology Made Simple,*** Simple Software Publishing, 1996) y Singer ***(Object Technology Strategies and Tactics,*** Cambndge University Press, 1996) son también valiosas introducciones a los conceptos y métodos OO.

Zamir ***(Handbook of Object Technology,*** CRC Press, 1998) ha editado un voluminoso tratado que cubre todos los aspectos de las tecnologías de objetos. Fayad y Laitnen ***(Trunsition to Object-Oriented Software Development,*** Wiley, 1998) utiliza casos de estudio para identificar los retos técnicos, culturales y de gestión a superar cuando una organización hace una transición a la tecnología de objetos. Gardner et al. ***(Cognitive Patterns: Problems-Solving Frameworks for Object Technology,*** Cambride University Press, 1998) proporcionar al lector una introducción básica sobre conceptos de resolución de problemas y la tecnología asociada a los patrones cognitivos y al modelado cognitivo tal y como se aplican en los sistemas orientados a objetos.

La naturaleza Única del paradigma OO supone especiales retos a los gestores de proyecto. Los libros de Cock-Burn ***(Surviving Object-Oriented Projects: A Manager's Guide,*** Addison-Wesley, 1998), Booch ***(Object Solutions: Managing the Object Oriented Project,*** Addison-Wesley, 1995), Goldberg y Rubin ***(Succeding With Objects: Decision Frameworks for Project Management***, Addison-Wesley, 1995) y Meyer ***(Object-Success: A Manager's Guide to Object Orientation,*** Prentice-Hall, 1995) consideran las estrategias de planificación, seguimiento y control de proyectos OO.

Earles y Simms ***(Building Business Objects,*** Wiley, 1998), Carmichael ***(Developing Business Objects,*** SIGS Books, 1998), Fingar ***(The Blueprint for Business Objects,*** Cambridge University Press, 1996) y Taylor ***(Business Engineerin with Object Technology,*** Wiley, 1995) enfocan la tecnología de objetos tal y como se aplica en contextos de negocios. Sus libros muestran métodos para expresar los conceptos y requisitos de negocio directamente como objetos y aplicaciones orientadas a objetos.

En Internet hay gran variedad de fuentes de información sobre tecnologías de objetos y otros temas relacionados. En **http://www.pressman5.com** encontrará una lista de referencias web actualizada sobre temas OO.

CAPÍTULO

21 ANÁLISIS ORIENTADO A OBJETOS

CUANDO se tiene que construir un nuevo producto o sistema, ¿cómo lo caracterizamos de forma tal que sea tratado por la ingeniería del software orientado a objetos? ¿Hay preguntas especiales que queramos hacer al cliente? ¿Cuáles son los objetos relevantes? ¿Cómo se relacionan entre **sí?** ¿Cómo se comportan los objetos en el contexto del sistema? ¿Cómo especificamos o modelamos un problema de forma tal que podamos crear un diseño eficaz?

A cada una de estas interrogantes se responde dentro del contexto del análisis orientado a objetos (AOO), primera actividad técnica que se desarrolla como parte de la ingeniería del software OO. En lugar de examinar un problema utilizando el modelo de información clásico de flujo de datos, el AOO introduce varios conceptos nuevos. Coad y Yourdon [COA91] consideran el tema cuando escriben:

> **El** AOO (Análisis Orientado a Objetos) se basa en conceptos que ya aprendimos en la guardería: objetos y atributos, clases y miembros, todos y partes. El porqué nos ha llevado tanto tiempo aplicar estos conceptos al análisis y especificación de sistemas es algo que nadie sabe a ciencia cierta**...**

El AOO se basa en un conjunto de principios básicos que se introdujeron en el Capítulo 11. Para construir un modelo de análisis se aplican cinco principios básicos: (1) se modela el dominio de la información; (2) se describe la función; **(3)** se representa el comportamiento del modelo; (4) los modelos de datos, funcional y de comportamiento se dividen para mostrar más detalles; y *(5)* los modelos iniciales representan la esencia del problema mientras que los Últimos aportan'detalles de la implementación. Estos principios forman la base para el enfoque del AOO presentado en ese capítulo.

VISTAZO RÁPIDO

¿Qué es? Antes de que pueda construir un sistema orientado a objetos, tiene que definir las clases (obietos) **que** representan el problema a resolver, la forma en que las clases se relacionan e interactúan unas **con** otras, el funcionamiento interno (atributos y operaciones) de los objetos y los mecanismos de comunicación (mensajes) que los permiten trabajar juntos. **Todas** estas **cosas** se cumplen en el análisis orientado a objetos.

¿Quién lo hace? La definición de un modelo de análisis lleva implícita una descripción de las características de las clases que describan un sistema o producto. La actividad la r e a l i i un ingeniero del software.

¿Por qué es importante? No se puede construir **software** (orientado a objetos o de otro tipo) hasta que *se* tiene un **conocimiento razonable del sistema o producto. El AOO nos proporciona una forma concreta de representar el conocimiento de los requisitos y una forma de probar dicho** conocimiento **enfrentándolo con la percepción que el cliente tiene del sistema** a **construir**.

¿Cuáles son los pasos? El AOO comienza con una descripción de casos de uso, que es **una** descripción de escenarios sobre cómo interactúan los actores (gente, máquinas **u otros** sistemas) con el sistema a construir. El modelo de **clases**-responsabilidad-colaboración **(CRC)** traslada la información **de los casos de uso a una representación de las clases y sus colaboraciones con las otras clases. Las características estáticas y dinámicas de las clases se modelan entonces utilizando un lenguaje de modelado unificado o cualquier otro método.**

¿Cuál es el producto obtenido? **Se crea un modelo de análisis orientado a objetos. Dicho modelo se compone de una representación gráfica, o basada en el lenguaje, que define los atributos** de la clase, las **relaciones y comportamientos y** las **comunicaciones entre** clases, así como una r**epresentación** del comportamiento de la clase en el tiempo.

El propósito del AOO es definir todas las clases que son relevantes al problema que se va a resolver, las operaciones y atributos asociados, las relaciones y comportamientos asociadas con ellas. Para cumplirlo se deben ejecutar las siguientes tareas:

1. Los requisitos básicos del usuario deben comunicarse entre el cliente y el ingeniero del software.
2. Identificar las clases (es decir, definir atributos y métodos).
3. Se debe especificar una jerarquía de clases.
4. Representan las relaciones objeto a objeto (conexiones de objetos).
5. Modelar el comportamiento del objeto.
6. Repetir iterativamente las tareas de la 1 a la 5 hasta completar el modelo.

Es importante observar que no existe un acuerdo universal sobre los «conceptos» que sirven de base para el AOO, pero un limitado número de ideas clave se repten a menudo, y son éstas las que consideraremos en este capítulo.

21.1 ANÁLISIS ORIENTADO A OBJETOS

El objetivo del análisis orientado a objetos es desarrollar una serie de modelos que describan el software de computadora al trabajar para satisfacer un conjunto de requisitos definidos por el cliente. El AOO, como los métodos de análisis convencional descritos en el Capítulo **12,** forman un modelo de análisis multiparte para satisfacer este objetivo. El modelo de análisis ilustra información, funcionamiento y comportamiento dentro del contexto de los elementos del modelo de objetos descrito en el Capítulo 20.

21.1.1. Enfoques convencionales y enfoques OO

¿Es el análisis orientado a objetos realmente diferente del enfoque del análisis estructurado presentado en el Capítulo 12? Aunque el debate continúa, Fichman y Kemerer [FIC92] responden a la pregunta directamente:

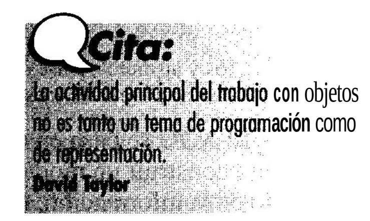
> Concluimos que el enfoque del análisis orientado a objetos... representa un cambio radical sobre las metodologías orientadas a procesos, tales como el análisis estructurado, pero **sólo** un cambio incremental respecto a las metodologías orientadas a datos, tales como la ingeniería de la información. Las metodologías orientadas a procesos desvían la atención de las prioridades inherentes a **los** objetos durante el proceso de modelado y conducen a un modelo del dominio del problema ortogonal con los tres principios esenciales de la orientación a objetos: encapsulamiento, clasificación de objetos y herencia.

Dicho simplemente, el análisis estructurado toma una visión diferente de los requisitos del modelo entrada-proceso-salida. **Los** datos se consideran separadamente de los procesos que los transforman. El comportamiento del sistema, aunque importante, tiende a desempeñar un papel secundario en el análisis estructurado. El análisis estructurado hace un fuerte uso de la descomposición funcional *(partición* del diagrama de flujo de datos, Capítulo 12).

21.1.2. El panorama del AOO

La popularidad de las tecnologías de objetos ha generado docenas de métodos de AOO desde finales de los **80** y durante los 90'. Cada uno de ellos introduce un proceso para el análisis de un producto o sistema, un conjunto de modelos que evoluciona fuera del proceso, y una notación que posibilita al ingeniero del software crear cada modelo de una manera consistente. Entre los más ampliamente utilizados se encuentran[1]:

El método de Booch. El método de Booch [BOO94] abarca un «microproceso de desarrollo» y un «macroproceso de desarrollo». El nivel micro define un conjunto de tareas de análisis que se reaplican en cada etapa en el macro proceso. Por esto se mantienen un enfoque evolutivo. El micro proceso de desarrollo identifica clases y objetos y la semántica de dichas clases y objetos, define las relaciones entre clases y objetos y realiza una serie de refinamientos para elaborar el modelo del análisis.



El método de Rumbaugh. Rumbaugh [RUM91] y sus colegas desarrollaron la *Técnica de Modelado de Objetos* (OMT) para el análisis, diseño del sistema y diseño a nivel de objetos. La actividad de análisis crea tres modelos: el modelo de objetos (una representación de objetos, clases, jerarquías y relaciones), el modelo dinámico (una representación del comportamiento del sistema y los objetos) y el modelo funcional (una representación a alto nivel del flujo de información a través del sistema similar al DFD).

El método de Jacobson. También llamado OOSE (en español Ingeniería del Software Orientada a Objetos), el método de Jacobson [JAC92] es una versión simplificada de Objectory, un método patentado, también desarrollado por Jacobson. Este método se diferencia de los otros por la importancia que da al *caso de uso* - una descripción o escenario que describe cómo el usuario interactúa con el producto o sistema —.

El método de Coad y Yourdon. El método de Coad y Yourdon [COA91] se considera, con frecuencia, como uno de los métodos del AOO más sencillos de aprender. La notación del modelado es relativamente simple y las reglas para desarrollar el modelo de análisis son evidentes. **A** continuación sigue una descripción resumida del proceso de AOO de Coad y Yourdon:

- Identificar objetos, usando el criterio de «qué buscar».
- Definir una estructura de generalización-especificación.

[1] La discusión detallada de estos métodos y sus diferencias esta fuera del alcance en este libro. Además, la industria se mueve hacia una forma de modelado unificada en un único método, lo que hace que las discusiones sobre los antiguos métodos sólo tengan sentido a efectos históricos. El lector interesado puede consultar a Berard [BER92] y Graham [GRA94] para una comparación más detallada.

[2] En general, los métodos de AOO se identifican usando el (o los) nombre(s) del desarrollador del método, incluso si al método se le ha dado un nombre o acrónimo único

- Definir una estructura de todo-parte.
- Identificar temas (representaciones de componentes de subsistemas).
- Definir atributos.
- Definir servicios.

El método de Wirfs-Brock. El método de Wirfs-Brock [WIR90] no hace una distinción clara entre las tareas de análisis y diseño. En su lugar, propone un proceso continuo que comienza con la valoración de una especificación del cliente y termina con el diseño. A continuación se esbozan brevemente las tareas relacionadas con el análisis de Wirfs-Brock:

- Evaluar la especificación del cliente.
- Usar un análisis gramatical para extraer clases candidatas de la especificación.
- Agrupar las clases en un intento de determinar superclases.
- Definir responsabilidades para cada clase.
- Asignar responsabilidades a cada clase.
- Identificar relaciones entre clases.
- Definir colaboraciones entre clases basándose en sus responsabilidades.
- Construir representaciones jerárquicas de clases para mostrar relaciones de herencia.
- Construir un grafo de colaboraciones para el sistema.

Aunque la terminología y etapas del proceso para cada uno de estos métodos de AOO difieren, los procesos generales de AOO son en realidad muy similares. Para realizar un análisis orientado a objetos, un ingeniero del software debería ejecutar las siguientes etapas genéricas:

1. Obtener los requisitos del cliente para el sistema.
2. Identificar escenarios o casos de uso.
3. Seleccionar clases y objetos usando los requisitos básicos como guías.
4. Identificar atributos y operaciones para cada objeto del sistema.
5. Definir estructuras y jerarquías que organicen las clases.
6. Construir un modelo objeto-relación.
7. Construir un modelo objeto-comportamiento.
8. Revisar el modelo de análisis OO con relación a los casos de uso/escenarios.

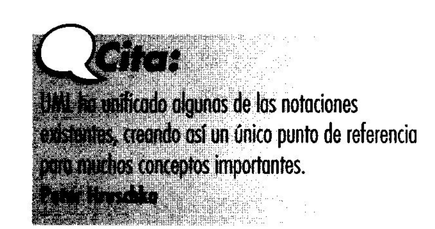
Estas etapas genéricas se tratan en mayor detalle en las Secciones 21.3 y 21.4.



Durante el AOO se aplico un conjunto genérico de pasos, independientemente del método de análisis elegido.

21.1.3. Un enfoque unificado para el AOO

Al final de la pasada década, Grady Booch, James Rumbaugh e Ivar Jacobson empezaron a colaborar para combinar y recopilar las mejores características de cada uno de sus métodos de diseño y análisis orientado a objetos en un método unificado. El resultado, denominado *Lenguaje de Modelado Unificado* (UML), se ha convertido en el método más utilizado por la industria[3].



UML permite a un ingeniero del software expresar un modelo de análisis utilizando una notación de modelado con unas reglas sintácticas, semánticas y prácticas. Eriksson y Penker [ERI98] definen dichas reglas de la siguiente forma:,

> La sintaxis nos dice cómo mostrar y combinar **los** símbolos. La sintaxis es comparable a las palabras en el lenguaje natural: es importante saber cómo se escriben y cómo combinarlas correctamente para formar una frase. Las regias semánticas nos dicen lo que significa cada símbolo y cómo interpretarlo, tanto cuando aparece **solo** como cuando lo hace en combinación con otros. Es comparable al significado de las palabras en el lenguaje natural.
>
> **Las** reglas prácticas definen el significado de **los** símbolos a través de los cuales se obtiene el modelo y se hace comprensible para otras personas. Esto correspondería, en lenguaje natural, a las regias de construcción de frases claras y fácilmente comprensibles.

En UML, un sistema viene representado por cinco vistas diferentes que lo describen desde diferentes perspectivas. Cada vista se representa mediante un conjunto de diagramas. En UML están presentes las siguientes vistas [ALH98]:

Vista del usuario. Representa el sistema (producto) desde la perspectiva de los usuarios (llamados actores en UML). El caso de uso es el enfoque elegido para modelar esta vista. Esta importante representación del análisis, que describe un escenario de uso desde la perspectiva del usuario final, se describió en el capítulo 11[4].

Vista estructural: **los** datos y la funcionalidad se muestran desde dentro del sistema, es decir, modela la estructura estática (clases, objetos y relaciones).



Como en todos los enfoques de análisis, lo obtención de requisitos es lo clave. Asegúrese de que obtiene lo vista correcta del usuario. El resto saldrá solo.

[3] Booch, Rumbaugh y Jacobson han publicado una trilogía fundamental de libros sobre **UML.** El lector interesado puede consultar [BOO99], [RUM99] y [JAC99].

[4] Si aún no lo ha hecho, lea por favor la discusión sobre los casos de uso de la Sección 11.2.4.

Vista del comportamiento: esta parte del modelo del análisis representa los aspectos dinámicos o de comportamiento del sistema. También muestra las interacciones o colaboraciones entre los diversos elementos estructurales descritos en las vistas anteriores.



En **mini.net/cetus/oo_uml.html** hay un amplio **tutorial** y una lista de recursos **UML** que incluye herramientas, **artículos** y ejemplos.

Vista de implementación: los aspectos estructurales y de comportamiento se representan aquí tal y como van a ser implementados.

Vista del entorno: aspectos estructurales y de comportamiento en el que el sistema a implementar se representa.

En general, el modelo de análisis de UML se centra en las vistas del usuario y estructural. El modelo de diseño de UML (tratado en el Capítulo 22) se dirige más a las vistas del comportamiento y del entorno. En el Capítulo 22 describiremos UML con más detalle.

21.2 ANÁLISIS DEL DOMINIO

El análisis en sistemas orientados a objetos puede ocurrir a muchos niveles diferentes de abstracción. A nivel de negocios o empresas, las técnicas asociadas con el AOO pueden acoplarse con un enfoque de ingeniería de la información (Capítulo 10) en un esfuerzo por definir clases, objetos, relaciones y comportamientos que modelen el negocio por completo. A nivel de áreas de negocios, puede definirse un modelo de objetos que describa el trabajo de un área específica de negocios (o una categoría de productos o sistemas). A nivel de las aplicaciones, el modelo de objetos se centra en los requisitos específicos del cliente, pues éstos afectan a la aplicación que se va a construir.



*El objetivo del **análisis** del dominio es definir el conjunto de clases **(objetos)** que se **encuentran** en el dominio de **la aplicación**. Dichas clases podrán reutilizarse **muchas** veces.*

El AOO, en su nivel de abstracción más alto (el nivel de empresa), está más allá del alcance de este libro. Los lectores interesados deberían ver a [EEL98], [CAR98], [FIN96], [MAT94], [SUL94] y {TAY95], los cuales hacen un análisis más detallado. A nivel de abstracción más bajo, el AOO cae dentro del alcance general de la ingeniería del software orientado a objetos y es el centro de atención del resto de las secciones de este capítulo. En esta sección nos centraremos en el AOO que se realiza a un nivel medio de abstracción. Esta actividad, llamada *análisis del dominio,* tiene lugar cuando una organización desea crear una biblioteca de clases reutilizables (componentes) ampliamente aplicables a una categoría completa de aplicaciones.

21.2.1. Análisis de reutilización y del dominio

Las tecnologías de objetos están influenciadas por la reutilización. Considere un ejemplo simple. El análisis de los requisitos para nuevas aplicaciones indican la necesidad de 100 clases. Se le asigna a dos equipos la tarea de construir la aplicación. Cada uno debe diseñar y construir un producto final y, a su vez, está compuesto de personas con el mismo nivel de habilidad y experiencia.



*Otros beneficios derivados de **la** reutilización **son** lo consistencia y **la familiaridad.** los patrones dentro del software serán más consistentes, tendiendo o **facilitar** el mantenimiento del producto. Asegúrese de establecer un conjunto de **reglas** de **reutilización** poro conseguir dichos objetivos.*

El equipo A no tiene acceso a una biblioteca de clases, por lo que debe desarrollar las 100 clases desde el principio. El equipo B usa una biblioteca de clases robusta y encuentra que ya existen 55 clases. Seguro que:

1. El equipo B finalizará el proyecto mucho antes que el A.
2. El coste del producto del equipo B será significativamente más bajo que el coste del producto del equipo A.
3. La versión que se distribuya del producto producido por el equipo B tendrá menos defectos que la del producto del equipo A.

Aunque el margen por el cual el trabajo del equipo B excederá al del A está abierto a debate, pocos argumentarán que la reusabilidad aporta una ventaja substancial al equipo B.

¿Pero de dónde vino la «biblioteca de clases robusta»? ¿Y cómo se determinó que las entradas de la biblioteca eran apropiadas para su uso en nuevas aplicaciones? Para responder estas preguntas, la organización que creó y mantuvo dicha biblioteca tuvo que aplicar el análisis del dominio.

21.2.2. El proceso de análisis del dominio

Firesmith [FIR93] describe el análisis del dominio del software de la siguiente manera:

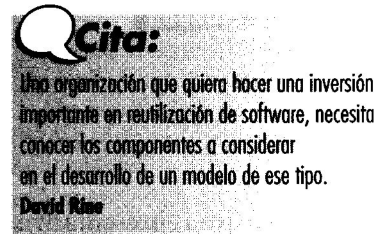
> **El análisis del dominio del software es la identificación, análisis y especificación de requisitos comunes de un dominio de aplicación específico, normalmente para su** *reutilización* **en múltiples proyectos dentro del mismo dominio de aplicación... (el análisis orientado a objetos del dominio es la identificación, análisis y especificación de capacidades comunes y** *reutilizables* **dentro de un dominio de aplicación específico, en términos de objetos, clases, submontajes y marcos de trabajo comunes...**

El «dominio de aplicaciones específico» puede variar desde aviónica hasta banca, desde juegos de vídeo multimedia hasta aplicaciones dentro de un escáner **CAT.** El objetivo del análisis del dominio es claro: encontrar o crear aquellas clases ampliamente aplicadas, de tal manera que sean reutilizables.

Referencia cruzada

La reutilización es la piedra angular de la ingeniería del software basada en componentes, tema que se aborda en el Capítulo 27.

Usando la terminología introducida al inicio del libro, el análisis del dominio puede verse como la actividad de cobertura para el proceso del software. Con esto queremos decir que el análisis del dominio es una actividad en curso de la ingeniería del software no ligada a ningún proyecto de software. En cierta forma, el papel de un análisis del dominio es similar al de un maestro tornero dentro de un entorno de fabricación fuerte. El trabajo del maestro tornero es diseñar y construir herramientas que pueden usarse por varias personas que trabajan en aplicaciones similares, pero no necesariamente idénticas. El papel del analista del dominio es diseñar y construir componentes reutilizables que puedan ser utilizados por diferentes personas que trabajan en aplicaciones similares pero no necesariamente iguales.



La Figura 21.1 [ARA89] ilustra las entradas y salidas clave para el proceso de análisis del dominio. Se examinan las fuentes del dominio de conocimiento en un intento de identificar objetos que se puedan reutilizar a lo largo de todo el dominio. En esencia el análisis del dominio es muy similar a la ingeniería del conocimiento. El ingeniero del conocimiento investiga un área de interés específica, intentando extraer hechos claves que se puedan usar para la construcción de un sistema experto o una red neurona1 artificial. Durante el análisis del dominio ocurre la extracción de objetos (y clases).

Definir el dominio a investigar. Para llevar a cabo esta tarea, el analista debe primero aislar el área de negocio, tipo de sistema o categoría del producto de interés. **A** continuación, se deben extraer los «elementos» tanto OO como no OO. Los elementos OO incluyen especificaciones, diseños y código para clases de aplicaciones OO ya existentes; clases de soporte (p.e.: clases de Interfaz Gráfica de Usuario o clases de acceso a bases de datos); bibliotecas de componentes comerciales ya desarrolladas (CYD) relevantes al dominio y casos de prueba. Los elementos no OO abarcan políticas, procedimientos, planes, estándares y guías; partes de aplicaciones no OO (incluyendo especificación, diseño e información de prueba), métricas y CYD del software no OO.

Clasificar los elementos extraídos del dominio. Los elementos se organizan en categorías y se establecen las características generales que definen la categoría. Se propone un esquema de clasificación para las categorías y se definen convenciones para la nomenclatura de cada elemento. Se establecen jerarquías de clasificación en caso de ser apropiado.

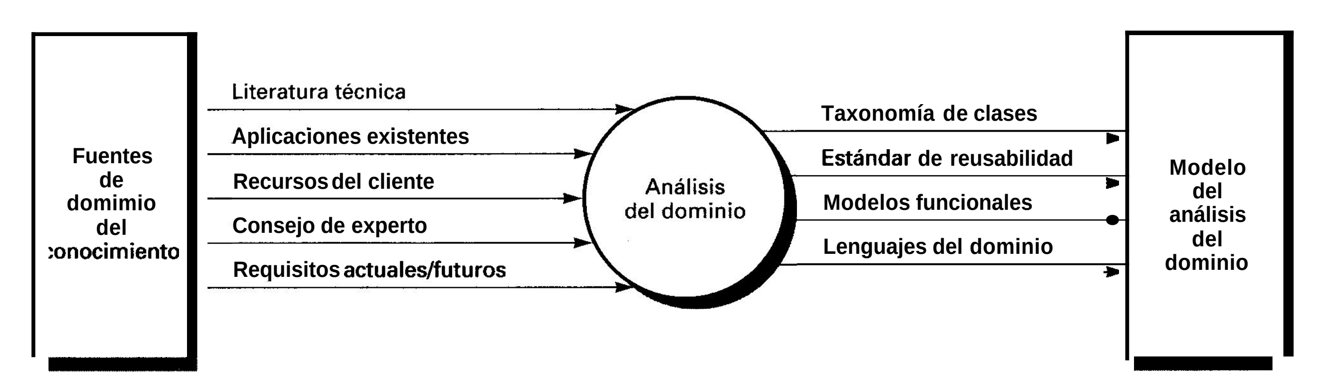


FIGURA 21.1. Entrada y salida para análisis del dominio.

Referencia cruzada

Una estrategia completa del modelo del análisis debe considerar tanto la arquitectura como los componentes. En el Capítulo 14 hay una completa discusión sobre la arquitectura del software.

Recolectar una muestra representativa de aplicaciones en el dominio. Para realizar esta tarea, el analista debe asegurar que la aplicación en cuestión tiene elementos que caen dentro de las categorías ya definidas. Berard [BER93] observa que durante las primeras etapas de uso de las tecnologías de objetos, una organización del software tendrá muy pocas, si hay alguna, aplicaciones OO. Debido a esto, el analista de dominio debe «identificar los objetos conceptuales (opuestos a los físicos) en cada aplicación [no OO]».

Analizar cada aplicación dentro de la muestra. El analista debe seguir las etapas siguientes [BER93]:

- Identificar objetos candidatos reutilizables.
- Indicar las razones que hacen que el objeto haya sido identificado como reutilizable.
- Definir adaptaciones al objeto que también pueden ser reutilizables.
- Estimar el porcentaje de aplicaciones en el dominio que pueden reutilizar el objeto.
- Identificar los objetos por nombre y usar técnicas de gestión de configuración para controlarlos (Capítulo 9).

Además, una vez que se han definido los objetos, el analista debe estimar qué porcentaje de una aplicación típica pudiera construirse usando los objetos reutilizables.

Desarrollar un modelo de análisis para los objetos. El modelo de análisis servirá como base para el diseño y construcción de los objetos del dominio.

Adicionalmente a estas etapas, el analista del dominio también debe crear un conjunto de líneas maestras para la reutilización, y desarrollar un ejemplo que ilustre cómo los objetos del dominio pudieran usarse para crear una aplicación nueva.

El análisis del dominio es la primera actividad técnica en una amplia disciplina que algunos llaman *ingeniería del dominio*. Cuando un negocio, sistema o producto del dominio es definido como estratégico a largo plazo, puede desarrollarse un esfuerzo continuado para crear una biblioteca reutilizable robusta. El objetivo es ser capaz de crear software dentro del dominio con un alto porcentaje de componentes reutilizables. Argumentos a favor de un esfuerzo dedicado a la ingeniería del dominio son: bajo coste, mayor calidad y menor tiempo de comercialización.



En **www.sei.cmu.edu/str/descriptions/deda.html** hay un tutorial sobre análisis del dominio que merece la pena.

21.3 COMPONENTES GENÉRICOS DEL MODELO DE ANÁLISIS OO

El proceso de análisis orientado a objetos se adapta a conceptos y principios básicos de análisis discutidos en el Capítulo 11. Aunque la terminología, notación y actividades difieren respecto de los usados en métodos convencionales, el AOO (en su núcleo) resuelve los mismos objetivos subyacentes. Rumbaugh [RUM91] examina esto cuando asegura que:

> El análisis... se ocupa de proyectar un modelo preciso, conciso, comprensible y correcto del mundo real. ...El propósito de análisis Orientado a objetos es modelar el mundo real de forma tal que sea comprensible. Para esto se deben examinar los requisitos, analizar las implicaciones que se deriven de ellos y reafirmarlos de manera rigurosa. Se deben abstraer primero las características del mundo real y dejar los pequeños detalles para más tarde.

Para desarrollar un «modelo preciso, conciso, comprensible y correcto del mundo real», un ingeniero del software debe seleccionar una notación que se soporte a un conjunto de componentes genéricos de AOO. Monarchi y Puhr [MON92] definen un conjunto de componentes de representación genéricos que aparecen en todos los modelos de análisis OO[5]. Los *componentes estáticos* son estructurales por naturaleza, e indican características que se mantienen durante toda la vida operativa de una aplicación. Los *componentes dinámicos* se centran en el control, y son sensibles al tiempo y al tratamiento de sucesos. Estos últimos definen cómo interactúa un objeto con otros a lo largo del tiempo. Pueden identificarse los siguientes componentes [MON92]:

Vista estática de clases semánticas. Una taxonomía de clases típicas se mostró en el Capítulo 20. Se imponen los requisitos y se extraen (y representan) las clases como parte del modelo de análisis. Estas clases persisten a través de todo el período de vida de la aplicación y se derivan basándose en la semántica de los requisitos del cliente.



[5] Los autores [MON92] aportan también un análisis de veintitrés métodos de AOO e indican cómo representan éstos dichas componentes.



Los componentes estáticos no cambian mientras la aplicación se está ejecutando. las componentes dinámicos están influenciados por el tiempo y los sucesos.

Vista estática de los atributos. Toda clase debe describirse explícitamente. Los atributos asociados con la clase aportan una descripción de la clase, así como una indicación inicial de las operaciones relevantes a esta clase.

Vista estática de las relaciones. Los objetos están «conectados» unos a otros de varias formas. El modelo de análisis debe representar las relaciones de manera tal que puedan identificarse las operaciones (que afecten a estas conexiones) y que pueda desarrollarse un buen diseño de intercambio de mensajes.

Vista estática de los comportamientos. Las relaciones indicadas anteriormente definen un conjunto de comportamientos que se adaptan al escenario utilizado (casos de uso) del sistema. Estos comportamientos se implementan a través de la definición de una secuencia de operaciones que los ejecutan.

Vista dinámica de la comunicación. Los objetos deben comunicarse unos con otros y hacerlo basándose en una serie de mensajes que provoquen transiciones de un estado a otro del sistema.

Vista dinámica del control y manejo del tiempo. Debe describirse la naturaleza y duración de los sucesos que provocan transiciones de estados.

De Champeaux y sus colegas [CHA93] definen una vista ligeramente diferente de las representaciones del **AOO.** Las componentes estáticas y dinámicas se identifican para el objeto internamente **y** para las representaciones entre objetos. Una vista dinámica e interna del objeto puede caracterizarse como la *historia de vida del objeto,* esto es, los estados que alcanza el objeto a lo largo del tiempo, al realizarse una serie de operaciones sobre sus atributos.

21.4 EL PROCESO DE AOO

El proceso de AOO no comienza con una preocupación por los objetos. Más bien comienza con una comprensión de la manera en la que se usará el sistema: por las personas, si el sistema es de interacción con el hombre; por otras máquinas, si el sistema está envuelto en un control de procesos; o por otros programas; si el sistema coordina y controla otras aplicaciones. Una vez que se ha definido el escenario, comienza el modelado del software.

Las secciones que siguen definen una serie de técnicas que pueden usarse para recopilar requisitos básicos del usuario y después definen un modelo de análisis para un sistema orientado a objetos.

21.4.1. Casos de uso

Como examinamos en el Capítulo 11, los casos de uso modelan el sistema desde el punto de vista del usuario. Creados durante la obtención de requisitos, los casos de uso deben cumplir los siguientes objetivos:

- definir los requisitos funcionales y operativos del sistema (producto), diseñando un escenario de uso acordado por el usuario final, **y** el equipo de desarrollo; proporcionar una descripción clara y sin ambigüedades de cómo el usuario final interactúa con el sistema y viceversa, y
- proporcionar una base para la validación de las pruebas.

Referencia cruzada

los casos de usa son una excelente herramienta de licitación de requisitos, independientemente del método de análisis utilizado. Véase el capítulo 11 para más detalle.

Durante el AOO los casos de uso sirven como base para los primeros elementos del modelo de análisis.

Utilizando UML se puede crear una representación visual de los casos de uso llamada diagrama de casos de uso. Como otros elementos del análisis, los casos de uso pueden representarse a diferentes niveles de abstracción. Los diagramas de casos de uso contienen casos de uso y actores, siendo estos últimos las entidades que interactúan con el sistema. Pueden ser humanos u otras máquinas o sistemas que tengan definidas interfaces con nuestro sistema.



En www.univ-paris1.fr/CRINFO/dmrg/MEE/misop014 existe un tutorial que aborda en profundidad los casos de uso.

El sistema de seguridad *HogarSeguro*, discutido en capítulos anteriores, puede usarse para ilustrar cómo pueden desarrollarse casos de uso. Recordando los requisitos básicos de *HogarSeguro*, podemos definir tres actores: el **propietario** (el usuario), los **sensores** (dispositivos adjuntos al sistema) y el **subsistema de monitorización y respuesta** (la estación central que monitoriza *HogarSeguro).* Para los propósitos de este ejemplo, consideraremos solamente el actor propietario.

La Figura 21.2.a muestra un diagrama de casos de uso de alto nivel para el **propietario.** En dicha figura

se identifican dos casos de uso y se representan con elipses. Cada caso de uso de alto nivel puede detallarse mediante diagramas de casos de uso de nivel inferior. Por ejemplo, la Figura 21.2.b representa un diagrama de casos de uso para la función *interactúa.* Se crea un conjunto completo de diagramas de casos de uso para todos los actores. Pero para una detallada discusión sobre los casos de uso usando UML es mejor consultar la bibliografía sobre el tema (p.e. [ERI97] o [ALH98]).

21.4.2. Modelado de clases-responsabilidades-colaboraciones

Una vez que se han desarrollado los escenarios de uso básicos para el sistema, es el momento de identificar las clases candidatas e indicar sus responsabilidades y colaboraciones. El modelado de *clases-responsabilidades-colaboraciones* (CRC) [WIR90] aporta un medio sencillo de identificar y organizar las clases que resulten relevantes al sistema o requisitos del producto. Ambler [AMB95] describe el modelado CRC de la siguiente manera:

> Un modelo **CRC** es realmente una colección de *tarjetas índice* estándar que representan clases. Las tarjetas están divididas en tres secciones. **A** lo largo de la cabecera de la tarjeta usted escribe el nombre de la clase. En el cuerpo se listan las responsabilidades de la clase a la izquierda y a la derecha **los** colaboradores.



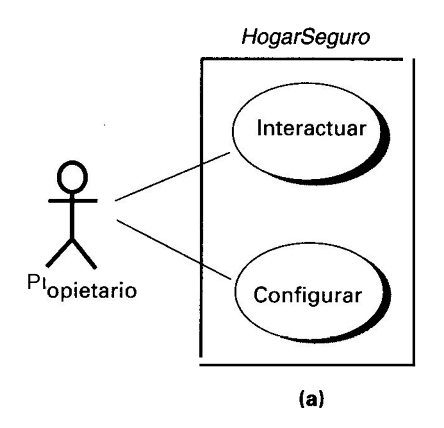
En **www.univ-paris1.fr/CRINFO/dmrg/MEE97/misop013** puede consultar una discusión detallada de la técnica de las tarjetas CRC.

En realidad, el modelo CRC puede hacer uso de tarjetas índice virtuales o reales. El caso es desarrollar una representación organizada de las clases. Las *responsabilidades* son los atributos y operaciones relevantes para la clase. Puesto de forma simple, una responsabilidad es «cualquier cosa que conoce o hace la clase» [AMB95]. Los *colaboradores* son aquellas clases necesarias para proveer a una clase con la información necesaria para completar una responsabilidad. En general, una colaboración implica una solicitud de información o una solicitud de alguna acción.

Clases

Las pautas básicas para identificar clases y objetos se presentaron en el Capítulo 20. Resumiendo, los objetos se manifiestan en una variedad de formas (Sección 20.3.1): entidades externas, cosas, ocurrencias o sucesos, roles, unidades organizativas, lugares, o estructuras. Una técnica para identificarlos en el contexto de un problema del software es realizar un análisis gramatical con la narrativa de procesamiento para el sistema. Todos los nombres se transforman en objetos potenciales. Sin embargo, no todo objeto potencial podrá incluirse en el modelo. Un objeto potencial debe satisfacer estas seis características para poder ser considerado como posible miembro del modelo:

1. *retener información:* el objeto potencial será útil durante el análisis si la información sobre el mismo debe guardarse para que el sistema funcione
2. *Servicios necesarios:* el potencial objeto debe tener un conjunto de operaciones identificables que permitan cambiar los valores de sus atributos.
3. *Múltiples atributos:* durante el análisis de requisitos nos centramos más en la información más importante. Un objeto con un solo atributo puede, en efecto, ser Útil durante el diseño, pero probablemente será un atributo de otro objeto durante el análisis de actividades.
4. *Atributos comunes:* el conjunto de atributos definido para la clase debe ser aplicable a todas las ocurrencias del objeto.



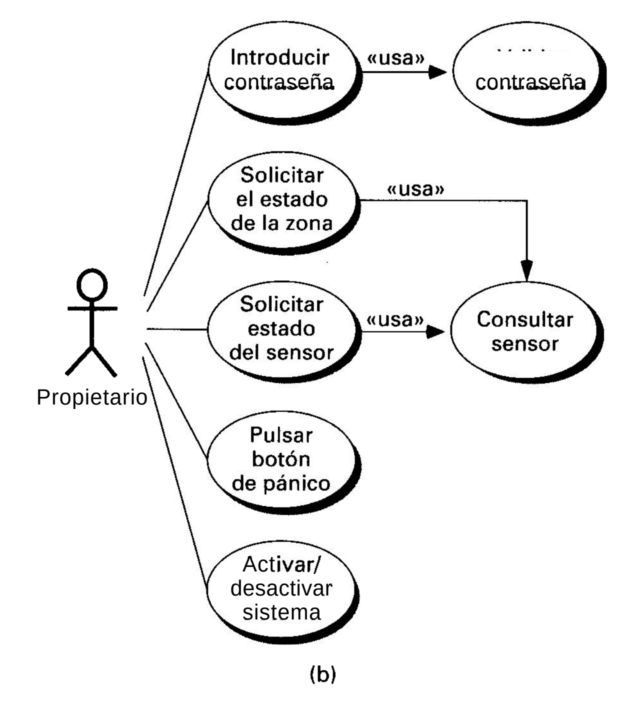
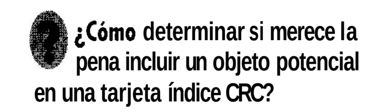


FIGURA 21.2. a) Diagrama de casos de **uso** de alto nivel. **b)** Diagrama de casos de uso detallado.


¿Cómo determinar si merece la pena incluir un objeto potencial en una tarjeta índice CRC?

5. *Operaciones comunes:* el objeto potencial debe definir un conjunto de operaciones aplicables, al igual que antes, a todos los objetos de la clase.
6. *Requisitos esenciales:* las entidades externas que aparecen en el espacio del problema y producen información esencial para la operación de una solución para el sistema casi siempre se definen como objetos en el modelo de requisitos.

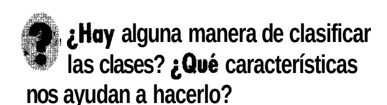
Una clase potencial debe satisfacer las seis características de selección anteriores si va a ser incluida en el modelo CRC.

Firesmith [FIR93] extiende las características de clasificación sugiriendo, además de las existentes, las siguientes:

Clases dispositivo. Modelan entidades externas tales como sensores, motores y teclados.

Clases propiedad. Representan alguna propiedad importante del entorno del problema (por ejemplo: establecimiento de créditos dentro del contexto de una aplicación de préstamos hipotecarios).

Clases interacción. Modelan interacciones que ocurren entre otros objetos (por ejemplo: una adquisición o una licencia).


¿Hay alguna manera de clasificar las clases? ¿Qué características nos ayudan a hacerlo?

Adicionalmente, los objetos y clases pueden clarificarse por un conjunto de características:

Tangibilidad. ¿Representa la clase algo tangible o palpable (por ejemplo, un teclado o sensor), o representa información más abstracta (por ejemplo: una salida prevista)?

Inclusividad. ¿Es la clase *atómica* (es decir, no incluye otras clases) o es *agregada* (incluye al menos un objeto anidado)?

Secuencia. ¿Es la clase *concurrente* (es decir posee su propio hilo de control) o *secuencial* (es controlada por recursos externos)?

Persistencia. La clase es *temporal* (se crea durante la ejecución del programa y es eliminada una vez que éste termina), o *permanente* (es almacenada en una base de datos)?

Integridad. ¿Es la clase *corrompible* (es decir, no protege sus recursos de influencias externas) o es *segura* (la clase refuerza los controles de accesos a sus recursos)?

Usando estas categorías de clases, pueden ampliarse las «tarjetas índice» creadas como parte del modelo CRC para incluir el tipo de la clase y sus características (Fig. 21.3).

Responsabilidades

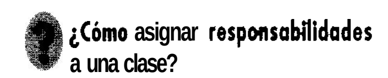
Las pautas básicas para identificar responsabilidades (atributos y operaciones) también fueron presentadas en el Capítulo 20. Para resumir, los atributos representan características estables de una clase, esto es, información sobre la clase que debe retenerse para llevar a cabo los objetivos del software especificados por el cliente. Los atributos pueden a menudo extraerse del planteamiento de alcance o discernirse a partir de la comprensión de la naturaleza de la clase. Las operaciones pueden extraerse desarrollando un análisis gramatical sobre la narrativa de procesamiento del sistema. Los verbos se transforman en candidatos a operaciones. Cada operación elegida para una clase exhibe un comportamiento de la clase.



Las responsabilidades de una clase incluyen tanto a **los** atributos como a las operaciones.

Wirfs-Brock y sus colegas [WIR90] sugieren cinco pautas para especificar responsabilidades para las clases:

1. *La inteligencia del sistema debe distribuirse de manera igualitaria.* Toda aplicación encierra un cierto grado de inteligencia, por ejemplo, lo que sabe el sistema y lo que puede hacer. Esta inteligencia puede distribuirse ente las clases de varias maneras. Las clases «tontas» (aquellas con pocas responsabilidades) pueden modelarse de manera que actúen como sirvientes de unas pocas clases «listas» (aquellas con muchas responsabilidades). Aunque este enfoque hace que el flujo de control dentro de un sistema sea claro, posee algunas desventajas: (1) Concentra toda la inteligencia en pocas clases, haciendo los cambios más difíciles; (2) Tiende a necesitar **más** clases y por lo tanto el esfuerzo de desarrollo aumenta.


¿Cómo asignar responsabilidades a una clase?

Por esta razón, la inteligencia del sistema debe distribuirse de manera igualitaria entre las clases de una aplicación. Como cada objeto conoce y actúa sobre algunos pocos elementos (generalmente bien definidos y claros), la cohesión del sistema se ve incrementada. Adicionalmente, los efectos colaterales provocados por cambios tienden a amortiguarse debido a que la inteligencia del sistema se ha descompuesto entre muchos objetos.



Si una clase tiene una lista excesivamente larga de responsabilidades, tal vez debería considerarse su división en varias clases menores.

Para determinar si la inteligencia del sistema está distribuida equitativamente, las responsabilidades definidas en cada tarjeta índice del modelo CRC deben ser evaluadas para determinar si cada clase posee una lista de responsabilidades extraordinariamente grande. Esto indica una concentración de inteligencia. Además, las responsabilidades de cada clase deben mostrar el mismo nivel de abstracción. Por ejemplo, entre la lista de operaciones de una clase agregada llamada **Comprobación de cuenta** existen **dos** responsabilidades: *saldo-de-la-cuenta* **y** *verificar-talones-cobrados*. La primera operación (responsabilidad) implica un complejo procedimiento matemático **y** lógico. La segunda es una simple actividad para empleados. Debido a que estas dos actividades no están al mismo nivel de abstracción, *verificar-talones-cobrados* debe situarse dentro de las responsabilidades de la clase **Comprobación de entradas,** la cual está contenida en la clase agregada **Comprobación de cuenta.**

Nombre de la clase:	
Tipo de la clase: (dispositivo, propiedad, rol, evento,...)	
Características de la clase: (tangible, atómica, concurrente,...)	
Responsabilidades:	**Colaboradores:**

FIGURA 21.3. Un modelo CRC de tarjeta índice.

2. *Cada responsabilidad debe establecerse lo más general posible*. Esta directriz implica que las responsabilidades generales (tanto los atributos como las operaciones) deben residir en la parte alta de la jerarquía de clases (puesto que son genéricas, se aplicarán a todas las subclases). Adicionalmente, debe usarse el polimorfismo (Capítulo 20) para definir las operaciones que generalmente se aplica a la superclase, pero que se implementan de manera diferente en cada una de las subclases.
3. *La información* y *el comportamiento asociado a ella, debe encontrarse dentro de la misma clase*. Esto implementa el principio OO de encapsulamiento (Capítulo 20). Los datos **y** procesos que manipulan estos datos deben empaquetarse como una unidad cohesionada.
4. *La información sobre un elemento debe estar localizada dentro de una clase, no distribuida a través de varias clases*. Una clase simple debe asumir la responsabilidad de almacenamiento **y** manipulación de un tipo específico de información. Esta responsabilidad no debe compartirse, de manera general, entre varias clases. Si la información está distribuida, el software se torna más difícil de mantener y probar.
5. *Compartir responsabilidades entre clases relacionadas cuando sea apropiado.* Existen muchos casos en los cuales una gran variedad de objetos exhibe el mismo comportamiento al mismo tiempo. Como un ejemplo, considere un videojuego que debe mostrar los siguientes objetos: **jugador, cuerpo-del-jugador, brazos-del-jugador, cabeza-del-jugador.** Cada uno de estos atributos tiene sus propios atributos (p.e.: *posición, orientación, color, velocidad),* y todos deben actualizarse **y** visualizarse al mover el usuario el *joystick*. Las responsabilidades *actualizar* **y** *visualizar* deben, por lo tanto, compartirse por los objetos señalados. El **jugador** sabe cuándo algo ha cambiado **y** se requiere *actualizar;* colabora con los otros objetos para alcanzar una nueva posición u orientación, pero cada objeto controla su propia *visualización.*

Colaboradores

Las clases cumplen con sus responsabilidades de una de las dos siguientes maneras: (1) una clase puede usar sus propias operaciones para manipular sus propios atributos, cumpliendo por lo tanto con una responsabilidad particular, o (2) puede colaborar con otras clases.

Wirfs-Brock [WIR90] **y** sus colegas definen las colaboraciones de la siguiente forma:

> Las colaboraciones representan solicitudes de un cliente a un servidor en el cumplimiento de una responsabilidad del cliente. Una colaboración es la realización de un contrato entre el cliente y el servidor. .. Decimos que un objeto colabora con otro, **si** para ejecutar una responsabilidad necesita enviar cualquier mensaje al otro objeto. Una colaboración simple fluye en una dirección, representando una solicitud del cliente al servidor. Desde el punto de vista del cliente, cada una de **sus** colaboraciones está asociada con una responsabilidad particular implementada por el servidor.



Un objeto servidor colabora con un objetos cliente en un esfuerzo por llevar o cabo una determinada responsabilidad. la colaboración implica el paso de mensajes.

Las colaboraciones identifican relaciones entre clases. Cuando todo un conjunto de clases colabora para satisfacer algún requisito, es posible organizarlas en un subsistema (un elemento del diseño).

Las colaboraciones se identifican determinando si una clase puede satisfacer cada responsabilidad. Si no puede, entonces necesita interactuar con otra clase. De aquí surge'lo que hemos llamado una colaboración.

Como un ejemplo, considere la aplicación *Hogar-Seguro*[6]. Como una parte del procedimiento de activación (vea el caso de uso para activación en la sección 11.2.4), el objeto **panel de control** debe determinar si existen sensores abiertos. Se define una responsabilidad llamada ***determinar-estado-del-sensor.*** Si hay sensores abiertos, el **panel de control** debe poner el atributo *estado* al valor «no preparado». La información del sensor puede obtenerse del objeto **sensor.** Por lo tanto, la responsabilidad ***determinar-estado-del-sensor*** puede ejecutarse solamente si el **panel de control** trabaja en colaboración con el **sensor.**

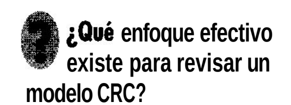
Para ayudar en la identificación de colaboradores, el analista puede examinar tres relaciones genéricas diferentes entre clases [WIR90]: (1) la relación *es-parte-de,* (2) la relación *tiene-conocimiento-sobre,* y *(3)* la relación *depende-de.* **A** través de la creación de un diagrama de relación entre clases (Sección 21.4.4), el analista desarrolla las conexiones necesarias para identificar estas relaciones. Cada una de las tres relaciones genéricas se considera brevemente en los párrafos siguientes.

Todas las clases que forman parte de una clase agregada están conectadas a ésta a través de una relación *es-parte-de.* Considere las clases definidas para el juego de vídeo mencionado anteriormente, la clase **cuerpo-del-jugador** *es-parte-de,* al igual que **cuerpo-del-jugador, brazos-del-jugador y cabeza-del-jugador.**

Cuando una clase debe obtener información sobre otra, se establece la relación *tiene-conocimiento-sobre.* La responsabilidad ***determinar-estado-del-sensor*** es un ejemplo de la relación *tiene-conocimiento-sobre.* La relación *depende-de* implica que dos clases poseen una dependencia no realizable a través de *tiene-conocimiento-sobre* o *es-parte-de.* Por ejemplo, la **cabeza-del-jugador** debe estar siempre conectada al **cuerpo-del-jugador** (a menos que el videojuego en particular sea muy violento), aunque cada objeto puede existir sin conocimiento directo sobre el otro. Un atributo del objeto **cabeza-del-jugador** llamado *posición-central* se obtiene de la *posición-central* del objeto **cuerpo-del-jugador.** Esta información se obtiene a través de un tercer objeto, jugador, el cual la obtiene del **cuerpo-del-jugador.** Por lo tanto, la **cabeza-del-jugador** *depende-de* **cuerpo-del-jugador.**

En todos los casos, el nombre de la clase colaboradora se registra en la tarjeta índice del modelo CRC al lado de la responsabilidad que ha generado dicha colaboración. Por lo tanto, la tarjeta índice contiene una lista de responsabilidades y de colaboraciones correspondientes que posibilitan se realicen las responsabilidades su realización (Fig. 21.3).

Cuando se ha desarrollado un modelo CRC por completo, los representantes del cliente y de las organizaciones de ingeniería del software pueden recorrer el modelo haciendo uso del siguiente enfoque.


¿Qué enfoque efectivo existe para revisar un modelo CRC?

1. **A** todos los participantes de la revisión (del modelo CRC) se les da un subconjunto de las tarjetas índice del modelo CRC. Las tarjetas que colaboran deben estar separadas (esto es, ningún revisor debe poseer dos tarjetas que colaboren).
2. Todos los escenarios (y sus correspondientes diagramas de casos de uso) deben organizarse en categorías.
3. El director de la revisión lee el caso de uso con atención. Cuando el director llega a un objeto identificado, se traspasa la señal a la persona que posee la clase tarjeta índice correspondiente. Por ejemplo, el caso de uso mencionado en la Sección 20.4.1 contiene la siguiente narración:

   > El propietario de la casa observa el **panel de control** de *Hogarseguro* para determinar si el sistema está listo para entrada de datos. Si el sistema *no está listo,* el propietario debe cerrar, físicamente, las ventanas y puertas para que aparezca el indicador de «listo». [Un indicador *no listo* implica que un sensor está abierto, esto es que una puerta o ventana está abierta.]

   Cuando el director de la revisión llega al «panel de control» en la narrativa del caso de uso, se pasa la señal a la persona que posee la tarjeta **panel de control.** La frase «implica que un sensor está abierto» requiere que la tarjeta índice contenga una responsabilidad que validará esta implicación (la responsabilidad ***determinar-estado-del-sensor*** realiza esta acción). **Al** lado de la responsabilidad en la tarjeta índice está el colaborador **sensor.** La señal se pasa entonces al objeto **sensor.**
4. Cuando se pasa la señal, se le pide al que posee la tarjeta de la clase que describa las responsabilidades mencionadas en la tarjeta. El grupo determina si una (o más) de las responsabilidades satisface el requisito del caso de uso.
5. **Si** las responsabilidades y colaboraciones mencionadas en las tarjetas índice no pueden acomodarse al caso de uso, se hacen modificaciones a las tarjetas. Esto puede incluir la definición de nuevas clases (y sus correspondientes tarjetas índice CRC) o la especificación de responsabilidades y colaboraciones nuevas o revisadas en tarjetas existentes.

Este *modus operandi* continúa hasta terminar el caso de uso. Cuando terminan todos los casos de uso, continúa el análisis OO.

21.4.3. Definición de estructuras y jerarquías

Una vez que se han identificado las clases y objetos usando el modelo CRC, el analista comienza a centrarse en la estructura del modelo de clases y las jerarquías resul-

[6] Vea una explicación de las clases de *Hogarseguro* en la Sección 20.3.

tantes que surgen al emerger clases y subclases. Usando la notación UML podemos crear gran variedad de diagramas. Debe crearse una *estructura de generalización-especialización* para las clases identificadas.

Para ilustrarlo considere el objeto **sensor** definido para *HogarSeguro* y mostrado en la Figura 21.4. **Aquí,** la generalización, **sensor,** se refina en un conjunto de especializaciones: **sensor de entrada, sensor de humo y sensor de movimiento.** Los atributos y operaciones identificados en el sensor son heredados por las especializaciones de la clase. Hemos creado una jerarquía de clases simple.

En otros casos, un objeto representado según el modelo inicial puede estar compuesto realmente de un número de partes las cuales pueden definirse a su vez como objetos. Estos objetos agregados pueden representarse como una estructura *componente-agregado* [ERI97] y se definen usando la notación representada en la Figura 21.5. El rombo implica una relación de ensamblaje. Debe notarse que las líneas de conexión pueden aumentarse con símbolos adicionales (no mostrados) para representar cardinalidad, que se adaptan de la notación del modelo entidad-relación descrito en el Capítulo 12.

Las representaciones estructurales proveen al analista de los medios para particionar el modelo CRC y para representar esta partición gráficamente. La expansión de cada clase aporta los detalles necesarios para revisión y para el subsiguiente diseño.

21.4.4. Definición de subsistemas

Un modelo de análisis para una aplicación compleja puede tener cientos de clases y docenas de estructuras. Por esta razón, es necesario definir una representación concisa que sea un resumen de los modelos CRC y estructural descritos anteriormente.



Un subsistema (paquete de UML) incluye una jerarquía de clases más detallada.

Los subconjuntos de clases que colaboran entre sí para llevar a cabo un conjunto de responsabilidades cohesionadas, se les llama normalmente *subsistemas* o *paquetes* (en terminología UML). Los subsistemas o paquetes son abstracciones que aportan una referencia o puntero a los detalles en el modelo de análisis. Si se observa desde el exterior, un subsistema puede tratarse como una caja negra que contiene un conjunto de responsabilidades y que posee sus propios colaboradores (externos). Un subsistema implementa uno o más *contratos* [WIR90] con sus colaboradores externos. Un contrato es una lista específica de solicitudes que los colaboradores pueden hacer a un subsistema[7].

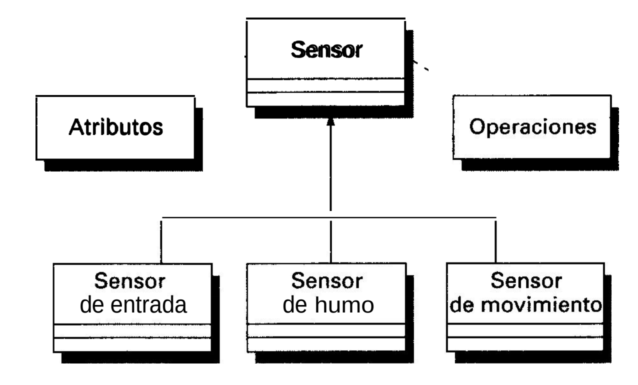


FIGURA 21.4.a Diagrama de clases que ilustra la relación de generalización-especialización.

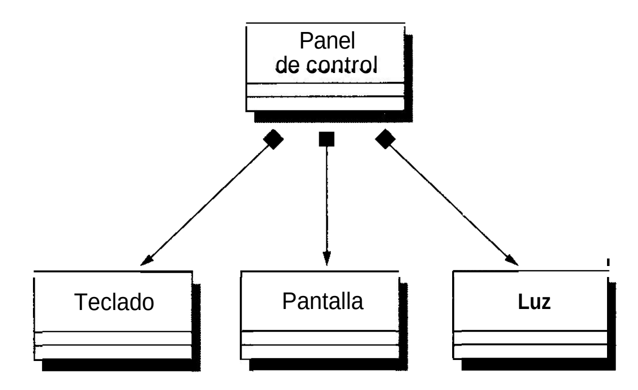


FIGURA 21.5.a Diagrama de clases que ilustra la relación de agregación.

Los subsistemas pueden representarse dentro del contexto del modelo CRC creando una tarjeta índice del subsistema. La tarjeta índice del subsistema indica el nombre del subsistema, los contratos que debe cumplir y las clases u otros subsistemas que soportan el contrato.

Los paquetes son idénticos a los subsistemas en intención y contenido, pero se representan gráficamente en UML. Por ejemplo, suponga que el panel de control de *HogarSeguro* es considerablemente más complejo que el representado por la Figura 21.5, conteniendo múltiples monitores, una sofisticada distribución de teclas y otras características. Éste pudiera modelarse como una estructura todo-partes mostrada en la Figura 21.6. Si el modelo de requisitos contuviera docenas de estas estructuras (*HogarSeguro* no las tendrá), sería difícil absorber la representación completa de una vez. Definiendo una referencia de

[7] Recuerde que las clases interactúan usando una filosofía cliente/servidor. En este caso el subsistema es el servidor y los colaboradores externos los clientes.

temas, como se muestra en la figura, pudiera referenciarse toda la estructura a través de un simple icono (la carpeta de ficheros). Las referencias de paquetes se crean generalmente para cualquier estructura que posea múltiples objetos.

Al nivel más abstracto, el modelo de AOO contendrá solamente referencias de paquetes tales como las que se ilustran al inicio de la Figura 21.7. Cada una de las referencias se expandirá a una estructura. Se ilustran las estructuras para los objetos **panel de control** y **sensor** (Figs. 21.5 y 21.6); para el **sistema, suceso sensor** y **alarma audible** se crearán también estructuras si estos objetos necesitan objetos ensamblados.

Las flechas discontinuas mostradas en la parte superior de la Figura 21.7 representan relaciones de dependencia entre los paquetes que se muestran. **Sensor** depende del estado del paquete **suceso sensor.** Las flechas continuas representan composición. En el ejemplo, el paquete **sistema** está compuesto por los paquetes **panel de control, sensor** y **alarma audible.**

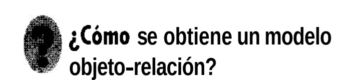
21.5 EL MODELO OBJETO-RELACIÓN

El enfoque del modelado CRC usada en secciones anteriores ha establecido los primeros elementos de las relaciones de clases y objetos. El primer paso en el establecimiento de las relaciones es comprender las responsabilidades de cada clase. La tarjeta índice del modelo CRC contiene una lista de responsabilidades. El siguiente paso es definir aquellas clases colaboradoras que ayudan en la realización de cada responsabilidad. Esto establece la «conexión» entre las clases.

Entre dos clases cualesquiera que estén conectadas existe una *relación*[8]. Debido a esto los colaboradores siempre están relacionados de alguna manera. El tipo de relación más común es la binaria (existe una relación entre dos clases). Cuando se analiza dentro del contexto de un sistema **OO,** una relación binaria posee una dirección específica[9] que se define a partir de qué clase desempeña el papel del cliente y cuál actúa como servidor.

Rumbaugh y sus colegas [RUM91] sugieren que las relaciones pueden derivarse a partir del examen de los verbos o frases verbales en el establecimiento del alcance o casos de uso para el sistema. Usando un análisis gramatical, el analista aísla verbos que indican localizaciones físicas o emplazamientos *(cerca de, parte de, contenido en),* comunicaciones *(transmite a, obtenido de),* propiedad *(incorporado por, se compone de)* y cumplimiento de una condición *(dirige, coordina, controla).* Estos aportan una indicación de la relación.

La notación del lenguaje unificado de modelado para el modelo objeto-relación utiliza una simbología adaptada de las técnicas del modelo entidad-relación examinadas en el Capítulo 12. En esencia, los objetos se conectan con otros objetos utilizando relaciones con nombres. Se especifica la cardinalidad de la conexión (ver capítulo 12) y se establece toda una red de relaciones.



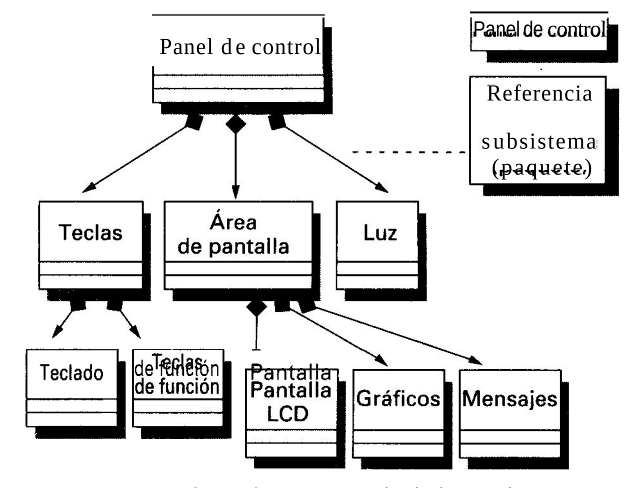


FIGURA 21.6. Referencia a paquete (subsistema).

El modelo objeto-relación (como el modelo entidad-relación) puede obtenerse en tres pasos o etapas:

1. *Usando las tarjetas índice CRC, puede dibujarse una red de objetos colaboradores.* La Figura 21.8 representa las conexiones de clase para los objetos de *HogarSeguro.* Primero se dibujan los objetos conectados por líneas sin etiquetas (no se muestran en la figura) que indican la existencia de alguna relación entre los objetos conectados. Una alternativa es mostrar los nombres de los roles de cada clase en la relación en lugar del nombre de la relación. Esto se describe en el Capítulo 22.
2. *Revisando el modelo CRC de tarjetas índices, se evalúan responsabilidades y colaboradores y cada línea de conexión sin etiquetar recibe un nombre.* Para evitar ambigüedades, una punta de flecha indica la «dirección» de la relación (Fig. 21.8).
3. *Una vez que se han establecido y nombrado las relaciones, se evalúa cada extremo para determinar* ***la*** *car-*

[8] Otros términos para relación son asociación [RUM91] y conexión [COA91].

[9] Es importante notar que esta es una salida de la naturaleza bidireccional de las relaciones usadas en el modelado de datos (Capítulo 12).

dinalidad (Fig. 21.8). Existen cuatro opciones: 0 a 1, 1 a 1, 0 a muchos, ó 1 a muchos. Por ejemplo, el sistema *HogarSeguro* contiene un único panel de control (la cardinalidad 1 lo indica). Al menos un sensor debe estar presente para que el panel de control lo active. Sin embargo, varios sensores pueden estar presentes (la relación 1..* lo indica). Un sensor puede reconocer 1 o más sucesos de sensor (p.e.: se detecta humo u ocurre una caída, pues el símbolo 1..* lo indica).

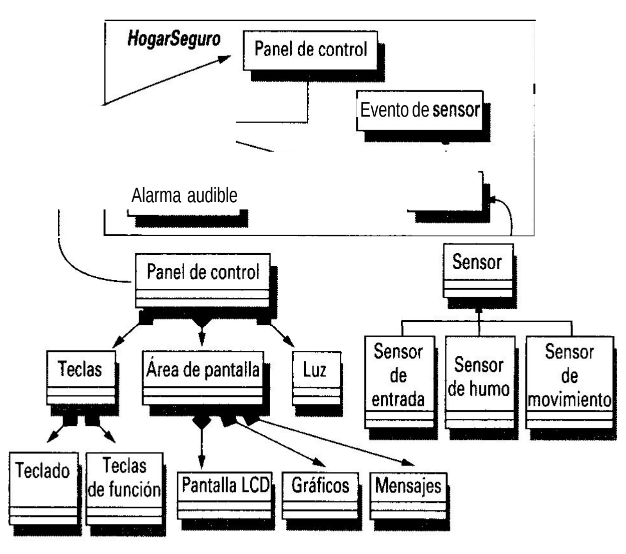


FIGURA 21.7. Modelo de análisis con referencias a paquetes.

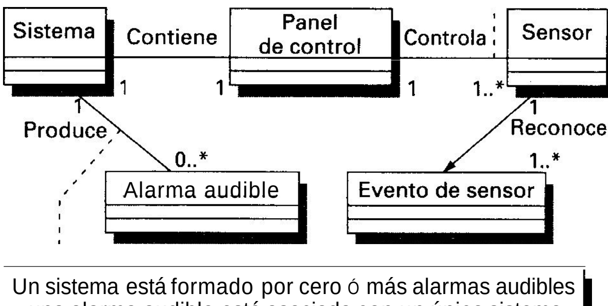


FIGURA 21.8. Relaciones entre objetos.

Los pasos anteriormente mostrados continúan hasta que se produzca un modelo objeto-relación completo.

En el desarrollo de un modelo objeto-relación, el analista añade aún alguna otra dimensión al modelo de análisis general. No solamente se identifican las relaciones entre objetos, sino que se definen todas las vías importantes de mensajes (Capítulo 20). En nuestra descripción de la Figura 21.7, hicimos referencia a las flechas que conectan símbolos de paquetes. También son vías de mensajes. Cada flecha implica el intercambio de mensajes entre subsistemas en el modelo.

21.6 EL MODELO OBJETO-COMPORTAMIENTO

El modelo CRC y el de objeto-relación representan elementos estáticos del modelo de análisis OO. Ahora es el momento para hacer una transición al comportamiento dinámico del sistema o producto OO. Para ejecutar este paso debemos representar el comportamiento del sistema como una función de sucesos específicos y tiempo.

¿Cuáles son los posos o seguir poro construir un modelo objetos-tomportomiento?

El modelo *objeto-comportamiento* indica cómo responderá un sistema OO a sucesos externos o estímulos. Para crear el modelo, el analista debe ejecutar los siguientes pasos:

1. Evaluar todos los casos de uso (Sección 21.4.1) para comprender totalmente la secuencia de interacción dentro del sistema.
2. Identificar sucesos que dirigen la secuencia de interacción y comprender cómo estos sucesos se relacionan con objetos específicos.
3. Crear una traza de sucesos [RUM91] para cada caso de uso.
4. Construir un diagrama de transición de estados para el sistema.
5. Revisar el modelo objeto-comportamiento para verificar exactitud y consistencia.

Cada uno de estos pasos se discute en las secciones siguientes.

21.6.1. Identificación de sucesos con casos de uso

Como vimos en la Sección 21.4.1, el caso de uso representa una secuencia de actividades que incluyen a actores y al sistema. En general, un *suceso* ocurre cada vez que un sistema OO y un actor (recuerde que un actor puede ser una persona, un dispositivo o incluso un sistema externo) intercambian información. Recordando la explicación dada en el Capítulo 12, es importante recordar que un suceso es Booleano. Esto es, un suceso *no* es la información que se intercambia, es el hecho de que la información ha sido intercambiada.

Un caso de uso se examina por puntos de intercambio de información. Para ilustrarlo, reconsidere el caso de uso descrito en la Sección 11.2.4:

1. El propietario observa el panel de control de *HogarSeguro* (Figura **11.2)** para determinar si el sistema está listo para recibir datos. Si el sistema no está listo, el propietario debe cerrar físicamente ventanas **y** puertas de tal manera que el indicador de disponibilidad esté presente. [Un indicador de

no preparado implica que el sensor está abierto, por ejemplo, que una puerta o ventana está abierta.]

2. El propietario usa el teclado para teclear una contraseña de cuatro dígitos. La contraseña se compara con la contraseña válida almacenada en el sistema. Si la contraseña es incorrecta, el panel de control avisará una vez y se restaurará por sí mismo para recibir datos adicionales. Si la contraseña es correcta, el panel de control espera por las acciones siguientes.
3. El propietario selecciona y teclea *permanecer* o *continuar* para activar el sistema. *Permanecer* activa solamente los sensores del perímetro (los sensores internos detectores de movimiento están desactivados). *Continuar* activa todos los sensores.
4. Cuando ocurre la activación, el propietario puede observar una luz roja de alarma.

Las partes de texto subrayadas anteriormente indican sucesos. Deberá identificarse un actor para cada suceso: la información que se intercambia debe anotarse. y deberán indicarse otras condiciones o restricciones.

Como ejemplo de un suceso típico, considere la frase subrayada del caso de uso propietario usa el teclado para teclear una contraseña de cuatro dígitos. En el contexto del modelo de análisis OO el actor **propietario** transmite un suceso al objeto **panel de control.**

El suceso puede llamarse *contraseña introducida.* La información transferida son los cuatro dígitos que forman la contraseña, pero ésta no es una parte esencial del modelo de comportamiento. Es importante notar que algunos sucesos tienen un impacto explícito en el flujo de control del caso de uso, mientras que otros no impactan directamente en este flujo de control. Por ejemplo. el suceso *contraseña introducida no* cambia explícitamente el flujo de control del caso de uso, pero los resultados del suceso *comparar contraseña* (derivada de la interacción contraseña se compara con la contraseña válida almacenada en el sistema) tendrá un impacto explícito en la información y flujo de control del software *HogarSeguro.*

Una vez que todos los sucesos han sido identificados, se asocian a los objetos incluidos. Los actores (entidades externas) y objetos pueden responsabilizarse de la generación de sucesos (p.e.: **propietario** genera el suceso *contraseña introducida)* o reconociendo sucesos que han ocurrido en otra parte (p.e.: el panel de control reconoce el resultado binario del suceso *comparar contraseña).*

21.6.2. Representaciones de estados

En el contexto de sistemas OO deben considerarse dos caracterizaciones de estados: (1) el estado de cada objeto cuando el sistema ejecuta su función, y (2) el estado del sistema observado desde el exterior cuando éste ejecuta su función.

El estado de un objeto adquiere en ambos casos características pasivas y activas [CHAR93]. Un estado pasivo es simplemente el estado actual de todos los atributos de un objeto. Por ejemplo, el estado pasivo del objeto agregado **jugador** (en la aplicación del videojuego examinado anteriormente) incluirá **posición** y **orientación** actual del **jugador** (atributos del objeto) así como otras características de jugador que son relevantes al juego (p.e.: un atributo que indique **permanecen deseos mágicos**). El *estado activo* de un objeto indica el estado actual cuando éste entra en una transformación continua o proceso. El objeto **jugador** poseerá los siguientes estados activos: *en movimiento, en descanso, lesionado, en recuperación, atrapado, perdido* entre otros. Para forzar la transición de un objeto de un estado activo a otro debe ocurrir un suceso (a veces, llamado *disparador).* Un componente de un modelo objeto-comportamiento es una representación simple de los estados activos de cada objeto y los sucesos (disparadores) que producen los cambios entre estos estados activos. La Figura 21.9 ilustra una representación simple de los estados activos para el objeto **panel de control** en el sistema *HogarSeguro.*



Cuando se empiezan a identificar estados, la atención se centra en los modos de comportamiento observables desde el exterior. Más tarde, se pueden refinar estos estados en comportamientos no evidentes cuando se observa el sistema desde el exterior.

Cada flecha en la Figura 21.9 representa una transición de un estado activo del objeto a otro. Las etiquetas mostradas en cada flecha representan los sucesos que disparan la transición. Aunque el modelo de estado activo aporta una visión interna muy útil de la «historia de vida» de un objeto, es posible especificar información adicional para aportar más profundidad en la comprensión del comportamiento de un objeto. Adicionalmente a especificar el suceso que provoca la ocurrencia de la transición, el análisis puede también especificar una guarda y una acción [CHAR93]. Una *guarda* es una condición Booleana, que debe satisfacerse para posibilitar la ocurrencia de una transición. Por ejemplo, la condición de guarda para la transición desde el estado «en descanso» al de «comparando» en la Figura 20.9 puede determinarse a través del examen del caso de uso:

if (entrada de contraseña = 4 dígitos)
then ejecutar transición al estado comparando;

En general, la guarda de una transición depende usualmente del valor de uno o más atributos de un objeto. En otras palabras, la condición de guarda depende del estado pasivo del objeto.

Una *acción* ocurre concurrentemente con la transición o como una consecuencia de ella y generalmente implica una o más operaciones (responsabilidades) del objeto. Por ejemplo, la acción conectada con el suceso *contraseña introducida* (Fig. 21.9) es una operación que accede a un objeto contraseña y realiza una comparación dígito a dígito para validar la contraseña introducida.

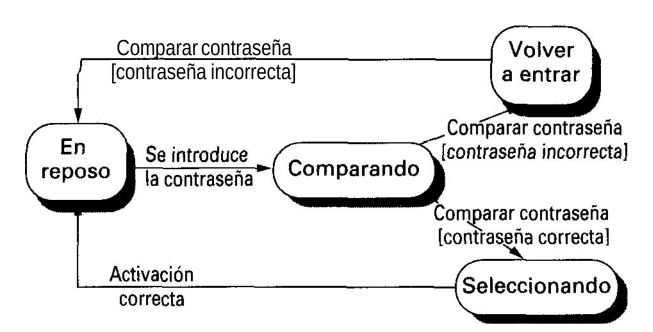


FIGURA 21.9. Representación de la transición de estados.

El segundo tipo de representación de comportamiento para el AOO considera una representación de estados para el producto general o sistema. Esta representación abarca un modelo simple de traza de sucesos [RUM91] que indica cómo los sucesos causan las transiciones de objeto a objeto y un diagrama de transición de estados que ilustra el comportamiento de cada objeto durante el procesamiento.

Una vez que han sido identificados los sucesos para un caso de uso, el analista crea una representación acerca de cómo los sucesos provocan el flujo desde un objeto a otro. Esta representación, llamada *traza de sucesos o de sucesos,* es una versión abreviada del caso de uso que representa objetos clave y los sucesos que provocan el comportamiento de pasar de objeto a objeto.

CLAVE

Una transición de un estado a otro requiere un suceso que la produzca. los sucesos son booleanos por naturaleza ya menudo ocurren cuando un objeto se comunica con otro.

La Figura 21.10 ilustra una traza parcial de sucesos para el sistema *HogarSeguro.* Cada una de las flechas representa un suceso (derivado de un caso de uso) e indica cómo el suceso sintoniza su comportamiento entre los objetos de *HogarSeguro.* El primer suceso, *sistema listo,* se deriva del entorno exterior y sintoniza el comportamiento al **propietario.** El propietario teclea una contraseña. El suceso inicia *aviso* y *aviso sonoro* indican cómo se canaliza el comportamiento si la contraseña no es válida. Una contraseña válida provoca un retomo al **propietario.** Los sucesos restantes y sus trazas siguen el comportamiento como cuando se activa o desactiva el sistema.

UML utiliza diagramas de estado, de secuencia, de colaboración y de actividades para representar el comportamiento dinámico de los objetos y las clases identificadas como parte del modelo del análisis.

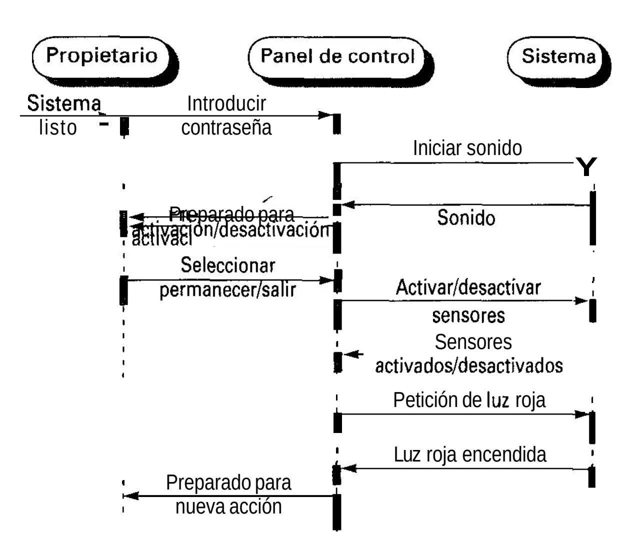


FIGURA 21.10. Traza de sucesos parcial para el sistema *Hogarseguro.*

RESUMEN

Los métodos de AOO permiten a un ingeniero del software modelar un problema representando las características tanto dinámicas como estáticas de las clases y sus relaciones como componentes principales del modelado. Como los métodos precedentes, el lenguaje unificado de modelado UML construye un modelo de análisis con las siguientes características: (1) representación de las clases y jerarquías de clases, (2) creación de modelos objeto-relación, y *(3)* obtención de modelos objeto-comportamiento.

El análisis de sistemas orientados a objetos se realiza a muchos niveles diferentes de abstracción. En los niveles de empresa o de negocio las técnicas asociadas con el análisis se pueden conjugar con el enfoque de ingeniería del proceso de negocio. A estas técnicas a menudo se las llama de análisis del dominio. En el nivel de implementación el modelo de objetos se centra en los requisitos especificados por el cliente tal y como éstos afectan a la aplicación a construir.

El proceso de AOO comienza con la definición de casos de uso (escenarios que describen cómo se va a utilizar el sistema). La técnica de modelado de clases-responsabilidades-colaboraciones (CRC) se aplica para documentar las clases y sus atributos y operaciones. También proporciona una vista inicial de las colaboraciones que ocurren entre los objetos. El siguiente paso en el AOO es la clasificación de objetos y la creación de una jerarquía de clases. Los sistemas (paquetes) se pueden utilizar para encapsular objetos relacionados. El modelo objeto-relación proporciona información sobre las conexiones entre las clases, mientras que el modelo objeto-comportamiento representa el comportamiento de los objetos individualmente y el global de todo el sistema.

REFERENCIAS

[ALH98] Alhir, S.S., *UML in a Nutshell,* O'Reilly & Associates, Inc. 1998.

[AMB95] Ambler, S., «Using Use-Cases», Software Development, Julio 1995, pp. 53-61

[ARA89] Arango, G., y Prieto-Diaz, R., «Domain Analysis: Concepts and Research Directions», *Domain Analysis: Acquisition of Reusable Information for Software Construction* (G. Arango y Prieto-Diaz, eds.), IEEE Computer Society Press, 1989.

[BER93] Berard, E.V., *Essays on Object-Oriented Software Engineering,* Addison-Wesley, 1993.

[BEN99] Beneth, S., S. McRobb y R. Farmer, *Object-Oriented System Analysis and Design Usign UNL,* McGraw-Hill, 1999.

[BOO94] Booch, G., *Object-Oriented Analysis and Design,* 2.ª ed., Benjamin Cummings, 1994.

[BOO99] Booch, G., Jacobson, I., y Rumbaugh, J., *The Unified Modeling Language User Guide,* Addison-Wesley, 1999.

[CAR98] Carmichael, A., *Developing Business Objects,* SIGS Books, 1998.

[CHA93] de Champeaux, D., D. Lea, y P. Faure, *Object-Oriented System Development,* Addison-Wesley, 1993.

[COA91] Coad, P., y E. Yourdon, *Object-Oriented Analysis,* 2.ª ed., PrenticeHall, 1991

[EEL98] Eeles, P,. y Simms, O., *Building Business Objects,* Wiley, 1998.

[ERI98] Eriksson, H.E., y Penker, M., *UML Toolkit,* Wiley, 1998.

[FIC92] Fichman, R.G., y Kemerer, C.F., *Object-Oriented and Conventional Analysis and Design Methodologies,* Computer, vol. 25, n.° 10, Octubre 1992, pp. 22-39.

[FIN96] Fingar, P., *The Blueprint for Business Objects,* Cambridge University Press, 1996.

[FIR93] Firesmith, D.G., *Object Oriented Requeriments Analysis and Logical Design,* Wiley, 1993.

[JAC92] Jacobson, I., *Object-Oriented Software Engineering,* Addison-Wesley, 1992.

[JAC99] Jacobson, I., Booch, G., y Rumbaugh, J., *Unified Software Development Process,* Addison-Wesley, 1999.

[MAT94] Mattison, R., *The Object-Oriented Enterprise,* McGraw Hill, 1994.

[MON92] Monarchi, D.E., y Puhr, G.I., «A Research Typology for Object-Oriented Analysis and Design», CACM, vol. 35, n.° 9, Septiembre 1992, pp. 35-47.

[RUM91] Rumbaugh, J., et al., *Object Oriented Modelling and Design,* Prentice-Hall, 1991.

[RUM99] Rumbaugh, J., Jacobson, I., y Booch, G., *The Unified Modelling Language Reference Manual,* Addison-Wesley, 1999.

[SUL94] Sullo, G.C., *Object Engineering,* Wiley, 1994.

[TAY95] Taylor, D.A., *Business Engineering with Object Technology*, Wiley, 1995.

[WIR90] Wirfs-Brock, R., Wilkerson, B., y Weiner, L., *Designing Object Oriented Software,* Prentice-Hall, 1990.

PROBLEMAS Y PUNTOS A CONSIDERAR

21.1. Hágase con uno o más libros sobre UML y compárelo con el análisis estructurado (Capítulo 12) utilizando las dimensiones de modelado propuestas por Fichman y Kemerer [FIC92] en la Sección 21.1.1.

21.2. Desarrolle una clase-presentación sobre un diagrama estático o dinámico de UML. Presente el diagrama en el contexto de un ejemplo sencillo, pero intente mostrar el suficiente nivel de detalle como para demostrar los aspectos más importantes del tipo de diagrama elegido.

21.3. Conduzca un análisis del dominio para una de las siguientes áreas:

a) Un sistema para almacenar los expedientes de los alumnos de una universidad.

b) Una aplicación de comercio electrónico (p.e. ropa, libros, equipos electrónicos, etc.)

c) Un servicio al cliente para un banco.

d) El desarrollo de un videojuego.

e) Un área de aplicación propuesta por su instructor.

21.4. Describa con sus propias palabras la diferencia entre las vistas estática y dinámica de un sistema.

21.5. Escriba un caso de uso para el sistema *HogarSeguro.* Los casos de uso deben tratar el escenario requerido para establecer una zona de seguridad. Una zona de seguridad lleva asociado un conjunto de sensores que pueden ser accedidos, activados y desactivados no individualmente sino en conjunto. Debe ser posible definir hasta diez zonas de seguridad. Sea creativo, pero intente mantenerse dentro de lo definido para el panel de control del sistema tal y como fue definido previamente en este libro.

21.6. Desarrolle un conjunto de casos de uso para el sistema SSRB del Problema 12.13.

Tendrá que hacer varias suposiciones sobre la forma de interacción del usuario con el sistema.

21.7. Desarrolle un conjunto de casos de uso para alguna de las siguientes aplicaciones:

a) Software para un asistente personal electrónico de propósito general.

b) Software para un videojuego de su elección.

c) Software para el sistema de climatización de un automóvil.

d) Software para un sistema de navegación de un automóvil.

e) Un sistema propuesto por su instructor.

Lleve a cabo una pequeña investigación (unas pocas horas) en el dominio de la aplicación y conduzca una reunión TFEA (Capítulo 11) con sus compañeros para desarrollar unos requisitos básicos (su instructor le ayudará a coordinarlo).

21.8. Desarrolle un conjunto completo de tarjetas CRC del producto o sistema elegido en el problema anterior.

21.9. Dirija una revisión de las tarjetas CRC con **sus** colegas. ¿Cuántas clases, responsabilidades y colaboraciones adicionales ha añadido a consecuencia de la reunión?

21.10. Desarrolle una jerarquía de clases para el producto o sistema elegido en el Problema 21.7.

21.11. Desarrolle un conjunto de subsistemas (paquetes) para el producto o sistema elegido en el Problema 21.7.

21.12. Desarrolle un modelo objeto-relación para el producto o sistema elegido en el Problema 21.7.

21.13. Desarrolle un modelo objeto-comportamiento para el producto o sistema elegido en el problema 21.7. Asegúrese **de** listar todos los sucesos, proporcionar la traza de sucesos, desarrollar un diagrama de flujo de sucesos y definir diagramas **de** estado para cada clase.

21.14. Describa con sus propias palabras la forma de determinar los colaboradores de una clase.

21.15. **¿Qué** estrategia propondría para definir subsistemas **en** colecciones de clases?

21.16. **¿Qué** papel desempeña la cardinalidad en el desarrollo de un modelo objeto-relación?

21.17. ¿Cuál es la diferencia entre los estados pasivo y activo de un objeto?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Los casos de uso forman la base del análisis orientado a objetos, sin importar el método de AOO elegido. Los libros de Rosemberg y Scott *(Use case driven Object modelling with UML: a practica1 approach,* Addison-Wesley, 1999), Schneider, Vinters y Jacobson *(Applying Use Cases: A Practica1 Guide,* Addison-Wesley, 1999), y Texel y Williams *(Use Cases Combined With Booch/OMT/UML: Process and Products,* Prentice-Hall, 1997) proporciona una guía para la creación y uso de esta importante herramienta de obtención de requisitos y mecanismo de representación.

Casi todos los libros publicados recientemente sobre análisis y diseño orientado a objetos ponderan UML. Todos los que están considerando en serio aplicar UML en su trabajo, deberían comprar [BOO99], [JAC99] y [RUM99]. Además de estos, los siguientes libros también son representativos de las docenas de ellos escritos sobre la tecnología de UML:

Douglas, B., y Booch, G., *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns,* Addison-Wesley, 1999.

Douglas, B., *Real-Time UML: Developing Efficient Objects for Embedded Systems,* Addison-Wesley, 1999.

Fowler, M., y Kendall Scott, *UML Distilled,* 2.ª ed., Addison-Wesley, 2000.

Larman, C., *Applying UML and Patterns: an Introduction to Object Oriented Analysis,* Prentice-Hall, 1997.

Odell, J.J., y Fowler, M., *Advanced Object Oriented Analysis and Design Using UML,* SIGS Books, 1998.

Ostereich, B., *Developing Software with UML: Object Oriented Analysis and Design in Practice,* Addison-Wesley, 1999.

Page-Jones, M., *Fundamentals of Object-Oriented Design in UML,* Addison-Wesley, 1999.

Stevens, P., y Pooley, R., *Software Engineering with Objects and Components,* Addison-Wesley, 1999.

En Intemet hay gran cantidad de fuentes de información sobre análisis orientado a objetos y temas relacionados. Una lista actualizada sobre las referencias web relevantes de cara al análisis orientado a objetos la encontrará en **http://www.pressman5.com**

CAPÍTULO

22 DISEÑO ORIENTADO A OBJETOS

EL diseño orientado a objetos transforma el modelo de análisis creado usando análisis orientado a objetos (Capítulo 21), en un modelo de diseño que sirve como anteproyecto para la construcción de software. El trabajo de diseñador de software puede ser intimidante. Gamma y sus colegas [GAM95] proveen un panorama razonablemente exacto del DOO cuando declaran que:

> El diseño de software orientado a objetos es difícil, y el diseño de software reusable orientado a objetos es aun más difícil. Se deben identificar los objetos pertinentes, clasificarlos dentro de las clases en la granularidad correcta, definir interfaces de clases y jerarquías de herencia y establecer relaciones clave entre ellos. El diseño debe ser específico al problema que se tiene entre manos, pero suficientemente general para adaptarse a problemas y requerimientos futuros. Además se deberá evitar el rediseño, o por lo menos minimizarlo. Los diseñadores experimentados en OO dicen que un diseño reusable y flexible es difícil, si no imposible de obtener «bien», la primera vez. Antes de que un diseño sea terminado, usualmente tratan de reutilizarlo varias veces, modificándolo cada vez.

A diferencia de los métodos de diseño de software convencionales, el DOO proporciona un diseño que alcanza diferentes niveles de modularidad. La mayoría de los componentes de un sistema, están organizados en subsistemas, un «módulo» a nivel del sistema. Los datos y las operaciones que manipulan los datos se encapsulan en objetos —una forma modular que es el bloque de construcción de un sistema OO—. Además, el DOO debe describir la organización específica de los datos de los atributos y el detalle procedural de cada operación. Estos representan datos y piezas algorítmicas de un sistema OO y son los que contribuyen a la modularidad global.

VISTAZO RÁPIDO

¿Qué es? El diseño de software Orientado a objetos requiere la definición de una arquitectura de software multicapa, la especificación de subsistemas que realizan funciones necesarias y proveen soporte de infraestructura, una descripción de objetos (clases), que son los bloques de construcción del sistema, y una descripción de los mecanismos de comunicación, que permiten que los datos fluyan entre las capas, subsistemas y objetos. El Diseño Orientado a Objetos (DOO), cumple todos estos requisitos.

¿Quién lo hace? El DOO lo realiza un ingeniero de software.

¿Por qué es importante? Un sistema orientado a objetos utiliza las definiciones de las clases derivadas del modelo de análisis. Algunas de estas definiciones tendrán que ser construidas desde el principio, pero muchas otras pueden ser reutilizadas, si se reconocen los patrones de diseño apropiados. El DOO establece un anteproyecto de diseño, que permite al ingeniero de software definir la arquitectura OO, en forma que maximice la reutilización; de esta manera, se mejora la velocidad del desarrollo y la calidad del producto terminado.

¿Cuáles son los pasos? El DOO se divide en dos grandes actividades: diseño del sistema y diseño de objetos. El diseño de sistema crea la arquitectura del producto definiendo una serie de «capas», que cumplen funciones específicas del sistema e identifica las clases, que son encapsuladas por los subsistemas que residen en cada capa.

Además, el diseño de sistemas incorpora la especificación de tres componentes: la interfaz de usuario, la gestión de *datos* y los mecanismos de administración de tareas. Ei diseño de objetos se centra en los detalles internos de cada clase, definición de atributos. operaciones y detalles de los mensqes.

¿Cuál es el producto obtenido? El modelo de diseño OO abarca arquitectura de software, descripción de la interfaz de usuario, componentes de gestión de datos. mecanismos de administración de tareas y descripciones detalladas de cada una de las clases usadas en el sistema.

¿Cómo puedo estar seguro de *que* lo he hecho correctamente? En cada etapa, los elementos del modelo de diseño orientado a objetos son revisados por claridad, corrección, integridad y consistencia con los requisitos del cliente y entre ellos.

La naturaleza Única del diseño orientado a objetos, reside en su capacidad para construir cuatro conceptos importantes de diseño de software: abstracción, ocultamiento (ocultación) de información, independencia funcional y modularidad (Capítulo 13). Todos los métodos de diseño procuran software que exhiba estas características fundamentales, pero sólo el DOO provee un mecanismo que permite al diseñador alcanzar las cuatro, sin complejidad ni compromiso.

El diseño orientado a objetos, la programación orientada a objetos, y las pruebas orientadas a objetos son actividades de construcción para sistemas OO. En este capítulo se considera el primer paso en la construcción.

22.1 DISEÑO PARA SISTEMAS ORIENTADOS A OBJETOS

En el Capítulo 13 se introdujo el concepto de una pirámide de diseño para el software convencional. Cuatro capas de diseño --datos, arquitectura, interfaz y nivel de componentes — fueron definidas y discutidas. Para sistemas orientados a objetos, podemos también definir una pirámide, pero las capas son un poco diferentes. Refiriéndose a la Figura 22.1, las cuatro capas de la pirámide de diseño OO son:

La capa subsistema. Contiene una representación de cada uno de los subsistemas, para permitir al software conseguir sus requisitos definidos por el cliente e implementar la infraestructura que soporte los requerimientos del cliente.

> **Cita:**
> En el diseño, configuramos el sistema y encontramos su forma ...
> **Ivor Jacobson, Grady Booch y James Rumbaugh.**

La capa de clases y objetos. Contiene la jerarquía de clases, que permiten al sistema ser creado usando generalizaciones y cada vez especializacionesmás acertadas. Esta capa también contiene representaciones.

La capa de mensajes. Contiene detalles de diseño, que permite a cada objeto comunicarse con sus colaboradores. Esta capa establece interfaces externas e internas para el sistema.

La capa de responsabilidades. Contiene estructuras de datos y diseños algorítmicos, para todos los atributos y operaciones de cada objeto.

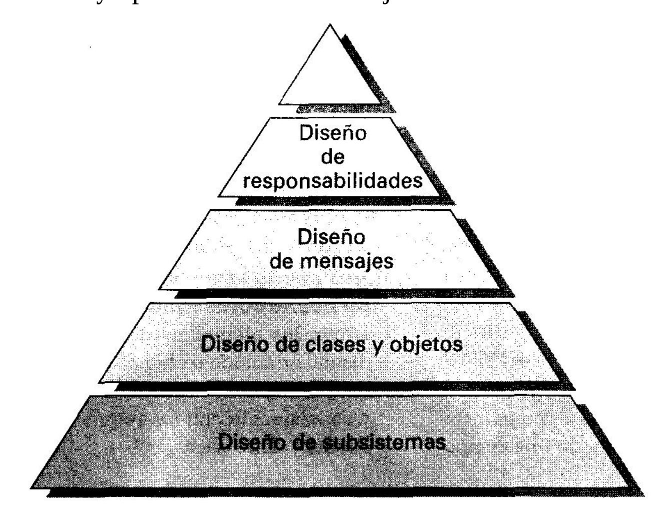


FIGURA 22.1. La pirámide del Diseño OO.

La pirámide de diseño se centia exclusivamente en el diseño de un producto o sistema específico. Observe, sin embargo, que existe otra capa de diseño, y que esta capa forma los cimientos sobre los que la pirámide se sostiene. La capa fundamental se centra en el diseño de los *objetos del dominio* (llamados *patrones de diseño*). Los objetos del dominio juegan un papel clave, en la construcción de la infraestructura del sistema OO aportando soporte para **las** actividades de interfaz hombre/máquina, administración (gestión) de tareas y gestión (administración) de datos. Los objetos del dominio se pueden usar, además, para desarrollar el diseño de la aplicación en **sí** misma.

22.1.1. Enfoque convencional vs. OO

Los enfoques convencionales para el diseño de software aplican distintas notaciones y conjunto de heurísticas, para trazar el modelo de análisis en un modelo de diseño. Recordando la Figura 13.1, cada elemento del modelo convencional de análisis se corresponde con uno o más capas del modelo de diseño. **Al** igual que el diseño convencional de software, el DOO aplica el diseño de datos cuando los atributos son representados, el diseño de interfaz cuando se desarrolla un modelo de mensajería, y diseño a nivel de componentes (procedimental), para operaciones de diseño. Es importante notar que la arquitectura de un diseño OO tiene más que ver con la colaboración entre objetos que con el control de flujo entre componentes del sistema.

A pesar de que existen similitudes entre los diseños convencionales y **OO,** se ha optado por renombrar las capas de la pirámide de diseño, para reflejar con mayor precisión la naturaleza de un diseño OO. La Figura 22.2 ilustra la relación entre el modelo de análisis OO (Capítulo 21) y el modelo de diseño que se derivará de ahí[1].

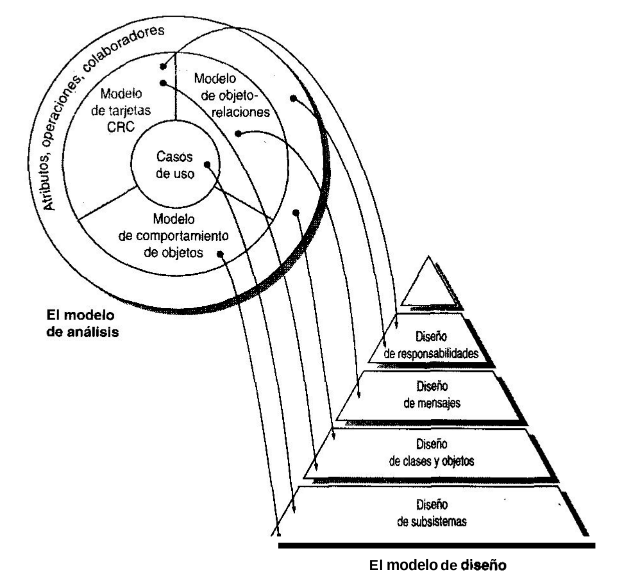


FIGURA 22.2. Transformación de un modelo de análisis OO a un modelo de diseño OO.

[1] Es importante hacer notar que la derivación no es siempre evidente. Para profundizar véase [DAV95].

El diseño de subsistemas se deriva considerando los requerimientos globales del cliente (representados por los casos de uso) y los sucesos y estados que son externamente observables (el modelo de comportamiento de objetos). El diseño de clases y objetos es trazado de la descripción de atributos, operaciones y colaboraciones contenidas en el modelo CRC. El diseño de mensajes es manejado por el modelo objeto-relación, y el diseño de responsabilidades es derivado del uso de atributos, operaciones y colaboraciones descrito en el modelo CRC.

Fichman y Kemerer [FIC92] sugieren diez componentes de diseño modelado, que pueden usarse para comparar varios métodos convencionales y orientados a objetos:

1. representación de la jerarquía de módulos.
2. especificación de las definiciones de datos.
3. especificación de la lógica *procedimental.*
4. indicación de secuencias de proceso final-a-final *(end-to-end)*
5. representación de estados y transiciones de los objetos.
6. definición de clases y jerarquías.
7. asignación de operaciones a las clases.
8. definición detallada de operaciones.
9. especificación de conexiones de mensajes.
10. identificación de servicios exclusivos.

¿Qué criterio se puede usar para comparar los métodos convencionales y los métodos de DOO?

Ya que existen muchos enfoques de diseño convencionales y orientados a objetos, es difícil desarrollar una comparación generalizada entre los dos métodos. Sin embargo, se puede asegurar que las componentes de modelado 5 al 10 no están soportadas usando diseño estructurado (Capítulo 14) o sus derivados.

22.1.2. Aspectos del diseño

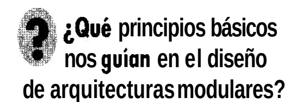
Bertrand Meyer [MEY90] sugiere cinco criterios para juzgar la capacidad de métodos de diseño para conseguir modularidad, y los relaciona al diseño orientado a objetos:

- *descomponibilidad:* la facilidad con que un método de diseño ayuda al diseñador a descomponer un problema grande en problemas más pequeños, haciéndolos más fácil de resolver.
- *componibilidad* el grado con el que un método de diseño asegura que los componentes del programa (módulos), una vez diseñados y construidos, pueden ser reutilizados para crear otros sistemas.
- *comprensibilidad* la facilidad con la que el componente de un programa puede ser entendido, sin hacer referencia a otra información o módulos.
- *continuidad:* la habilidad para hacer pequeños cambios en un programa y que se revelen haciendo los cambios pertinentes en uno o muy pocos módulos.
- *protección:* una característica arquitectónica, que reduce la propagación de efectos colaterales, si ocurre un error en un módulo dado.



Una referencia que responde a la pregunta ¿qué hace que un diseño orientado o objetos sea bueno?, puede encontrarse en: **www.kinetica.com/ootips/ood-principles.html**

De estos criterios, Meyer [MEY90], sugiere cinco principios básicos de diseño, que pueden ser deducidos para arquitecturas modulares: (1) unidades lingüísticas modulares, (2) pocas interfaces, *(3)* pequeñas interfaces (acoplamiento débil), (4) interfaces explícitas y *(5)* ocultación de información.



Los módulos se definen como unidades lingüísticas modulares, cuando «corresponden a unidades sintácticas en el lenguaje usado» [MEY90]. Es decir, el lenguaje de programación que se usará debe ser capaz de definir directamente la modularidad. Por ejemplo, si el diseñador crea una subrutina, cualquiera de los lenguajes de programación antiguos (FORTRAN, C, Pascal), debe poder implementarlos como una unidad sintáctica. Pero si un paquete que contiene estructuras de datos y procedimientos, y los identifica como una sola unidad definida, en un lenguaje como Ada (u otro lenguaje orientado a objetos), será necesario representar directamente este tipo de componente en la sintaxis del lenguaje.

Para lograr el bajo acoplamiento (un concepto de diseño introducido en el Capítulo 13), el número de interfaces entre módulos debe minimizarse («pocas interfaces»), y la cantidad de información que se mueve a través de la interfaz también debe ser minimizada («pequeñas interfaces»). Siempre que los componentes se comunican, deben hacerlo de manera obvia **y** directa («interfaces explícitas»). Por ejemplo, si el componente X y el componente Y se comunican mediante el área de datos global (a lo que se llama acoplamiento común en el Capítulo 13), violan el principio de interfaces explícitas porque la comunicación entre componentes no es obvia a un observador externo. Finalmente, se logra el principio de ocultamiento de información, cuando toda información acerca de un componente se oculta al acceso externo, a menos que esa información sea específicamente definida como *pública.*

Los criterios y principios de diseño presentados en esta sección pueden ser aplicados a cualquier método de diseño (así como al diseño estructurado). Como se verá, el método de diseño orientado a objetos logra cada uno de los criterios más eficientemente que otros enfoques, y resulta en arquitecturas modulares, que cumplen efectivamente cada uno de los criterios.

22.1.3. El Panorama de DOO

Como se vio en el Capitulo 21, una gran variedad de métodos de análisis y diseño orientados a objetos fue propuesta y utilizada durante los ochenta y los noventa. Estos métodos establecieron los fundamentos para la notación moderna de DOO, heurísticas de diseño y modelos. A continuación, haremos una breve revisión global de los primeros métodos de DOO:

El método de Booch. Como se vio en el Capítulo 21, el método Booch [BOO94] abarca un «proceso de micro desarrollo» y un «proceso de macro desarrollo». En el contexto del diseño, el macro desarrollo engloba una actividad de planificación arquitectónica, que agrupa objetos similares en particiones arquitectónicas separadas, capas de objetos por nivel de abstracción, identifica situaciones relevantes, crea un prototipo de diseño y valida el prototipo aplicándolo a situaciones de uso. El micro desarrollo define un conjunto de «reglas» que regulan el uso de operaciones y atributos y las políticas del dominio específico para la administración de la memoria, manejo de errores y otras funciones; desarrolla situaciones que describen la semántica de las reglas y políticas; crea un prototipo para cada política; instrumenta y refina el prototipo; y revisa cada política para así «transmitir su visión arquitectónica» [BOO94].

Cita:
No existe razón por la que la transición de la fase de requisitos a la de diseño no debería ser más fácil en la ingeniería de software que en cualquier otra disciplina de ingeniería. El diseño es difícil.
Alan Davis.

El método de Rumbaugh. La técnica de modelado de objetos (TMO) [RUM91] engloba una actividad de diseño que alienta al diseño a ser conducido a dos diferentes niveles de abstracción. El *diseño de sistema* se centra en el esquema de los componentes que se necesitan para construir un sistema o producto completo. El modelo de análisis se divide en subsistemas, los cuales se asignan a procesadores y tareas. Se define una estrategia para implementar la administración de datos, y se identifican los recursos y mecanismos de control requeridos para accesarlos. El diseño de objetos enfatiza el esquema detallado de un objeto individual. Se seleccionan las operaciones del modelo de análisis, y los algoritmos se definen para cada operación. Se representan las estructuras de datos apropiadas para atributos y algoritmos. Las clases y atributos de clase son diseñados de manera que se optimice el acceso a los datos, y se mejore la eficiencia computacional. Se crea un modelo de mensajería, para implementar relaciones de objetos (asociaciones).

El método de Jacobson. El diseño para ISOO (Ingeniería del software orientada a objetos) [JAC92] es una versión simplificada del método propietario *Objectory*, también desarrollado por Jacobson. El modelo de diseño enfatiza la planificación para el modelo de análisis ISOO. En principio, el modelo idealizado de análisis se adapta para acoplarse al ambiente del mundo real. Después los objetos de diseño primarios, llamados *bloques*[2], son creados y catalogados como bloques de interfaz, bloques de entidades y bloques de control. La comunicación entre bloques durante la ejecución se define y los bloques se organizan en subsistemas.

El método de Coad y Yourdon. Éste método para DOO [COA91], se desarrolló estudiando «cómo es que los diseñadores orientados a objetos efectivos» hacen su trabajo. La aproximación de diseño dirige no sólo la aplicación, sino también la infraestructura de la aplicación, y se enfoca en la representación de cuatro componentes mayores de sistemas: la componente de dominio del problema, la componente de interacción humana, la componente de administración de tareas y la de administración de datos.

El método de Wirfs-Brock. Wirfs-Brock, Wilkerson y Weiner [WIR90] definen un conjunto de tareas técnicas, en la cual el análisis conduce sin duda al diseño. Los *protocolos*[3] para cada clase se construyen refinando contratos entre objetos. Cada operación (responsabilidad) y protocolo (diseño de interfaz) se diseña hasta un nivel de detalle que guiará la implementación. Se desarrollan las especificaciones para cada clase (definir responsabilidades privadas y detalles de operaciones) y cada subsistema (identificar las clases encapsuladas y la interacción entre subsistemas).



Aunque no es tan robusto como UML, el método Wirfs-Brock tiene uno elegancia sencilla, que lo convierte en un enfoque alternativo e interesante al DOO.

A pesar de que la terminología y etapas de proceso para cada uno de estos métodos de DOO difieren, los procesos de DOO global son bastante consistentes.

[2] Un *bloque* es la abstracción de diseño, que permite la representación de un objeto agregado.

[3] Un *protocolo* es una descripción formal de los mensajes, a los que la clase responde.

Para llevar a cabo un diseño orientado a objetos, un ingeniero de software debe ejecutar las siguientes etapas generales:

1. Describir cada subsistema y asignar a procesadores y tareas.
2. Elegir una estrategia para implementar la administración de datos, soporte de interfaz y administración de tareas.
3. Diseñar un mecanismo de control, para el sistema apropiado.
4. Diseñar objetos creando una representación procedural para cada operación, y estructuras de datos para los atributos de clase.
5. Diseñar mensajes, usando la colaboración entre objetos y relaciones.
6. Crear el modelo de mensajería.
7. Revisar el modelo de diseño y renovarlo cada vez que se requiera.

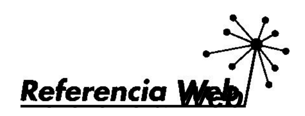


Un conjunto de etapas genéricas se aplica durante el DOO, sin importar el método de diseño que se escoja.

Es importante hacer notar que las etapas de diseño discutidas en esta sección son iterativas. Eso significa que deben ser ejecutadas incrementalmente, junto con las actividades de AOO, hasta que se produzca el diseño completo.

22.1.4. Un enfoque unificado para el DOO

En el Capítulo 21, se mencionó como Grady Booch, James Rumbaugh e Ivar Jacobson, combinaron las mejores cualidades de sus métodos personales de análisis y diseño orientado a objetos, en un método unificado. El resultado, llamado el *Lenguaje de Modelado Unificado,* se ha vuelto ampliamente usado en la industria[4].



Un tutorial y listado extenso de recursos de UML incluyendo herramientas, referencias y ejemplos, se pueden encontrar en: **mini.net/cetus/oo_uml.html**

Durante el modelo de análisis (Capítulo 21), se representan las vistas del modelo de usuario y estructural. Estas proporcionaron una visión interna al uso de las situaciones para el sistema (facilitando guías para el modelado de comportamiento), y establecieron fundamentos para la implementación y vistas del modelo ambiental, identificando y describiendo elementos estructurales estáticos del sistema.

UML se organiza en dos actividades mayores: diseño del sistema y diseño de objetos. El principal objetivo de UML, diseño de sistema, es representar la arquitectura de software. Bennett, Mc Robb y Farmer [BEN99], exponen este aspecto de la siguiente manera:

> En términos de desarrollo orientado a objetos, la arquitectura conceptual está relacionada con la estructura del modelo estático de clase y las conexiones entre las componentes del modelo. El modelo arquitectura describe la manera como el sistema se divide en subsistemas o módulos, y como se comunican exportando e importando datos. La arquitectura de código, define como es que el código del programa se encuentra organizado en archivos y directorios y agrupado en librerías. La arquitectura de ejecución se centra en los aspectos dinámicos del sistema y la comunicación entre componentes, mientras las tareas y operaciones se ejecutan.

La definición de «subsistemas», nombrada por Bennett et al., es una preocupación principal durante el *diseño de sistema* de UML.



El diseño de sistema se centra en arquitectura de software y definición de subsistemas. El diseño de objetos describe objetos, hasta un nivel en el cual puedan ser implementados, en un lenguaje de programación.

El *diseño de objetos* se centra en la descripción de objetos y sus interacciones con los otros. Una especificación detallada de las estructuras de datos de los atributos y diseño procedural de todas las operaciones, se crea durante el diseño de objetos. La *visibilidad*[5] para todos los atributos de clase se define, y las interfaces entre objetos se elaboran para definir los detalles de un modelo completo de mensajes.

El diseño de sistemas y objetos en UML se extiende para considerar el diseño de interfaces, administración de datos con el sistema que se va a construir y administración de tareas para los subsistemas que se han especificado. La interfaz de usuario en UML utiliza los mismos conceptos y principios examinados en el Capítulo 15. La visión del modelo de usua-

[4] Booch, Rumbaugh y Jacobson han escrito tres libros muy importantes sobre UML. El lector interesado debe consultar [BOO99], [RUM99] y [JAC99]

[5] Visibilidad indica si el atributo es público (disponible a través de todas las instancias de la clase), privado (disponible sólo para la clase que lo especifica) o protegido (un atributo que puede ser usado por la clase que lo especifica y por sus subclases).

rio maneja el proceso del diseño de la interfaz de usuario, proporcionando una situación que se elabora iterativamente, para volverse un conjunto de clases de interfaz[6]. La administración de datos establece un conjunto de clases y colaboraciones, que permiten al sistema (producto) manejar datos persistentes (por ejemplo, archivos y bases de datos). El diseño de la administración de tareas establece la infraestructura que organiza subsistemas en tareas, y administra la concurrencia de tareas. El flujo de procesos para el diseño se ilustra en la Figura 22.3'.

A lo largo del proceso de diseño UML, la visión del modelo de usuario y de estructura se elabora dentro del diseño de la representación delimitada anteriormente. Esta actividad de elaboración se analiza en las secciones siguientes.

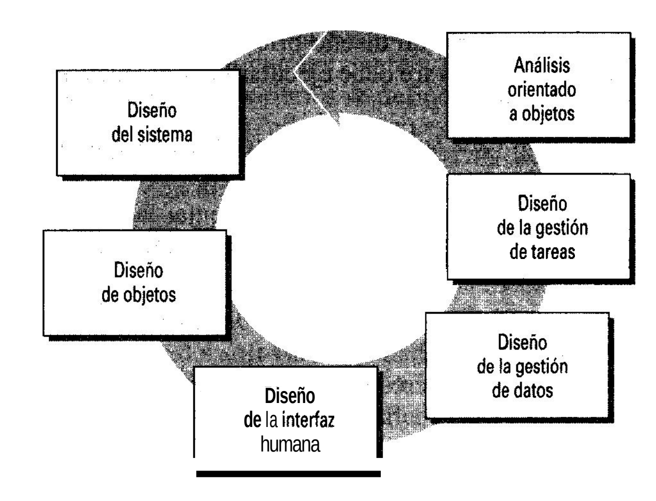
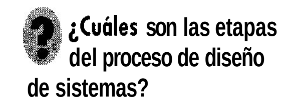


FIGURA 22.3. Flujo de Proceso **para** DOO.

22.2 EL PROCESO DE DISEÑO DE SISTEMA

El diseño de sistema desarrolla el detalle arquitectónico requerido para construir un sistema o producto. El proceso de diseño del sistema abarca las siguientes actividades:

- Partición del modelo de análisis en subsistemas.
- Identificar la concurrencia dictada por el problema.
- Asignar subsistemas a procesadores y tareas.
- Desarrollar un diseño para la interfaz de usuario.
- Elegir una estrategia básica para implementar la administración (gestión) de datos.
- Identificar recursos globales y los mecanismos de control requeridos para su acceso.



- Diseñar un mecanismo de control apropiado para el sistema, incluyendo administración de tareas.
- Considerar cómo deben manejarse las condiciones de frontera.
- Revisar y considerar *trade-offs*.

En las secciones siguientes, el diseño de actividades relacionadas con cada una de estas etapas se consideran con mayor detalle.

22.2.1. Particionar el modelo de análisis

Uno de los principios fundamentales del análisis (Capítulo 11) es hacer particiones. En el diseño de sistemas OO particiona el modelo de análisis, para definir colecciones congruentes de clases, relaciones y comportamiento. Estos elementos de diseño se definen como subsistema.



las conceptos de cohesión y acoplamiento (Capítulo 131 pueden aplicarse o nivel de subsistemas. Esfuércese par alcanzar uno buena independencia funcional, cuando diseñe subsistemas

En general, todos los elementos de un subsistema comparten alguna propiedad en común. Y se integran para completar la misma función; deben residir dentro del mismo producto de hardware, o deben administrar la misma clase de recursos. Los subsistemas se caracterizan por sus responsabilidades; eso significa que un subsistema puede identificarse por los servicios que provee [RUM91]. Cuando se usa en el contexto de un diseño de sistema **OO**, un servicio es una colección de operaciones que llevan a cabo una función específica (por ejemplo, administrar archivos de procesador de textos, producir un *rendering* tridimensional, traducir una señal de vídeo analógica en una imagen digital comprimida).

¿Qué criterios nos guían en el diseño de subsistemas?

Cuando se definen (y diseñan) los subsistemas, se deben seguir los siguientes criterios de diseño:

- El subsistema debe tener una interfaz bien definida, a través de la cual se reduzcan todas las comunicaciones con el resto del sistema.

[6] Hoy en día la mayoría de las clases de interfaz son parte de una librería de componentes de software reutilizables. Esto facilita el diseño e implementación de IGUs (Interfaz gráfica de usuario).

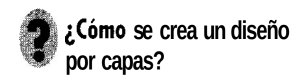
[7] Recuerde que el AOO es una actividad iterativa. Es totalmente posible que el modelo de análisis sea revisado como consecuencia del trabajo de diseño.

- Con la excepción de un pequeño número de «clases de comunicación», las clases incluidas dentro del subsistema deben colaborar sólo con otras clases dentro del subsistema.
- El número de subsistemas debe ser bajo.
- Un subsistema puede ser particionado internamente, para ayudar a reducir la complejidad.

Cuando dos subsistemas se comunican entre **sí,** pueden establecer un *enlace clientelservidor* o un enlace *punto-a-punto* (*peer-to-peer*) [RUM91]. En un enlace cliente/servidor, cada uno de los subsistemas asume uno de los papeles implicados, el de el cliente o el del servidor. El servicio fluye del servidor al cliente en una sola dirección. En un enlace punto-a-punto, los servicios pueden fluir en cualquier dirección.

Cuando un sistema es particionado en subsistemas, se lleva a cabo otra actividad de diseño, llamada estratificación por capas. Cada capa [BUS96] de un sistema **OO,** contiene uno o más subsistemas y representa un nivel diferente de abstracción de la funcionalidad requerida para completar las funciones del sistema. En la mayoría de los casos, los niveles de abstracción se determinan por el grado en que el procesamiento asociado con el subsistema es visible al usuario final.

Por ejemplo, una arquitectura de cuatro capas debe incluir: (1) una capa de presentación (el subsistema asociado con la interfaz de usuario), (2) una capa de aplicación (el subsistema que lleva a cabo procesos asociados con la aplicación), **(3)** una capa de formato de datos (los subsistemas que preparan los datos para ser procesados), y (4) una capa de base de datos (el subsistema asociado con la administración de datos). Cada capa se encuentra más profundamente dentro del sistema, representando un procesamiento más específico al ambiente.



Buschmann y sus colegas [BUS96] sugieren el siguiente enfoque de diseño para estratificación por capas:

1. Establecer los criterios de estratificación por capas. Esto significa decidir cómo se agruparán los subsistemas en una arquitectura de capas.
2. Determinar el número de capas. Muchas de ellas complican innecesariamente; muy pocas debilitan la independencia funcional.
3. Nombrar las capas y asignar subsistemas (con sus clases encapsuladas) a una capa. Asegurarse de que la comunicación entre subsistemas (clases) en una capa[8], y otros subsistemas (clases) en otra capa, siguen la filosofía de diseño de la arquitectura.
4. Diseñar interfaces para cada capa.
5. Refinar los subsistemas, para establecer la estructura de clases para cada capa.
6. Definir el modelo de mensajería para la comunicación entre capas.
7. Revisar el diseño de capas, para asegurar que el acoplamiento entre capas se minimiza (un protocolo cliente/servidor puede ayudar a realizar esta tarea)
8. Iterar para refinar el diseño de capas.

22.2.2. Asignación de concurrencia y subsistemas

El aspecto dinámico del modelo objeto-comportamiento provee una indicación de concurrencia entre clases (o subsistemas). Si las clases (o subsistemas) no se activan al mismo tiempo, no hay necesidad para el procesamiento concurrente. Esto significa que las clases (o subsistemas) pueden ser implementadas en el mismo procesador de hardware. Por otro lado, si las clases (o subsistemas) deben actuar en sucesos asincrónicamente y al mismo tiempo, se verán como concurrentes. Cuando los subsistemas son concurrentes, existen dos opciones de alojamiento: (1) alojar cada subsistema en procesadores independientes ó (2) alojar los subsistemas en el mismo procesador y proporcionar soporte de concurrencia, sobre las características del sistema operativo.



En la mayoría de los casas, una implementación de multiproceso incrementa la complejidad y el riesgo técnica. Siempre que sea posible, escoja lo arquitectura de procesador más simple que pueda realizar el trabajo.

Las tareas concurrentes se definen [RUM91] examinando el diagrama de estado para cada objeto. Si el flujo de sucesos y transiciones indica que solo un objeto está activo en el tiempo, un hilo de control se ha establecido. El hilo de control continúa, aun cuando un objeto envía un mensaje a otro objeto, mientras que el primer objeto espera por la respuesta. Sin embargo, si el primer objeto continúa procesando después de enviar un mensaje, el hilo de control se divide.

Las tareas en un sistema OO se diseñan generando hilos de control aislados. Por ejemplo, mientras que el sistema de seguridad *HogarSeguro* monitoriza sus sensores, puede también marcar a la estación central de monitorización para verificar la conexión. Ya que los objetos involucrados en ambos comportamientos están activos al mismo tiempo, cada uno representa un hilo de control y cada uno puede ser definido como una tarea distinta. Si la monitorización y marcado ocurrieran secuencialmente, podría implementarse una sola tarea.

Para determinar cuál de las opciones de asignación de procesadores es apropiada, el diseñador debe considerar los requisitos de desempeño, costos y el encabezado impuesto por la comunicación entre procesadores.

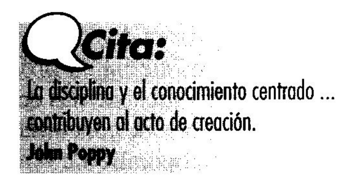
[8] En una arquitectura *cerrada*, los mensajes procedentes de una capa se deberían haber enviado a la capa adyacente inferior. En una arquitectura *abierta*, los mensajes deben enviarse a cualquiera de las capas inferiores.

22.2.3. Componente de administración de tareas

Coad y Yourdon [COA91] sugieren la estrategia siguiente, para el diseño de objetos que manipulan tareas concurrentes:

- se determinan las características de la tarea.
- se define un coordinador de tarea y objetos asociados.
- se integra el coordinador y otras tareas.

Las características de la tarea se determinan, comprendiendo cómo es que se inicia la tarea. Las tareas controladas por sucesos y manejadas por reloj son las más comunes. Ambas se activan por una interrupción, pero la primera recibe una interrupción de alguna fuente externa (por ejemplo, otro procesador, un Sensor), mientras que la última es controlada por el reloj.



Además de la manera en que una tarea es iniciada, también se deben determinar la prioridad y criticidad de la tarea. Las tareas de alta prioridad deben tener acceso inmediato a los recursos del sistema. Las tareas de alta criticidad deben continuar operando aun cuando la disponibilidad de un recurso es reducida o el sistema operativo se encuentra en estado degradado.

Una vez que las características de la tarea se han determinado, se definen los atributos y operaciones del objeto requerido, para alcanzar coordinación y comunicación con otras tareas. La plantilla básica de una tarea (para un objeto tarea), toma la forma de [COA91]:

Nombre de la tarea – el nombre del objeto
Descripción – un relato que describe el propósito del objeto.
Prioridad – prioridad de la tarea (por ejemplo, alta, media, baja).
Servicios – lista de operaciones que son responsabilidad del objeto.
Coordinados por – la manera como se invoca el comportamiento del objeto.
Comunicados vía – datos de entrada y salida relevantes a la tarea.

La descripción de esta plantilla puede ser traducida en el modelo de diseño estándar (incorporando la representación de atributos y operaciones), para los objetos tarea.

22.2.4. Componente de interfaz de usuario

Aunque la componente de interfaz de usuario se implementa dentro del contexto del dominio del problema, la interfaz por sí misma representa un subsistema de importancia crítica para la mayoría de las aplicaciones modernas. El modelo de análisis OO (Capítulo **21**), contiene los escenarios de uso (llamados ***casos** de uso*), **y** una descripción de los roles que juegan los usuarios (llamados ***actores)*** cuando interactúan con el sistema. Este modelo sirve como entrada al proceso de diseño de interfaz de usuario.

Referencia cruzada
La mayoría de los clases necesarias para crear una interfaz moderna ya existen y están disponibles para el diseñador. El diseño de interfaz obedece al enfoque definida en el Capítulo 15.

Una vez que el actor y su situación de uso se definen se identifica una jerarquía de comando. La jerarquía de órdenes define la mayoría de las categorías del menú de sistema (la barra de menú o la paleta de herramientas), **y** todas las subfunciones, que estarán disponibles en el contexto de una categoría importante de menú de sistema (las ventanas de menú). La jerarquía de órdenes se refina repetidamente, hasta que cada caso de uso pueda ser implementado navegando por la jerarquía de funciones.

Debido a que existe una amplia variedad de entornos de desarrollo de interfaces de usuario, el diseño de los elementos de una IGU (Interfaz Gráfica de Usuario) no es necesario. Ya existen clases reutilizables (con atributos y operaciones apropiadas) para ventanas, iconos, operaciones de ratón y una amplia gama de otro tipo de funciones de interacción. La persona que implementa estas clases (el *desarrollador*), sólo necesita ***instanciar*** objetos, con las características apropiadas para el dominio del problema.

22.2.5. Componente de la administración de datos

La administración o gestión de datos engloba dos áreas distintas de interés: (1) la administración (gestión) de datos críticos para la propia aplicación, y **(2)** la creación de infraestructura para el almacenamiento y recuperación de los objetos. En general, la administración de datos se diseña en forma de capas. La idea es aislar los requisitos de bajo nivel que manipulan las estructuras de datos, de los requisitos de alto nivel para manejar los atributos del sistema.

En el contexto del sistema, un sistema de manipulación de bases de datos, normalmente se usa como almacén de datos común para todos los subsistemas. Los objetos requeridos para manipular la base de datos son miembros de clases reutilizables que se identifican mediante el análisis del dominio (Capítulo **21**), o que se proporcionan directamente por el fabricante de la base de datos. Una discusión detallada del diseño de

bases de datos para sistemas OO está fuera del ámbito de este libro[9].

El diseño de la componente de administración de datos incluye el diseño de los atributos y operaciones requeridas para administrar objetos. Los atributos significativos se añaden a cada objeto en el dominio del problema, y proporcionan información que responde a la pregunta «¿Cómo me almaceno?». Coad y Yourdon [COA91] aconsejan la creación de una clase objeto-servidor, «con los servicios de (a) indicar al objeto que se almacene a sí mismo, y (b) recuperar objetos almacenados para su uso por otros componentes de diseño».

Como un ejemplo de la gestión de los datos para el objeto **Sensor,** examinado como parte del sistema de seguridad *HogarSeguro*, el diseño puede especificar un archivo llamado «Sensor». Cada registro debería corresponder a una instancia denominada **Sensor,** y habría de contener los valores de cada atributo de **Sensor** para una instancia dada. Las operaciones dentro de la clase objeto-servidor deberían permitir a un objeto específico ser almacenado y recuperado, cuando sea requerido por el sistema. Para objetos más complejos, sería necesario especificar una base de datos relacional, o una base de datos orientada a objetos para ejecutar la misma función.

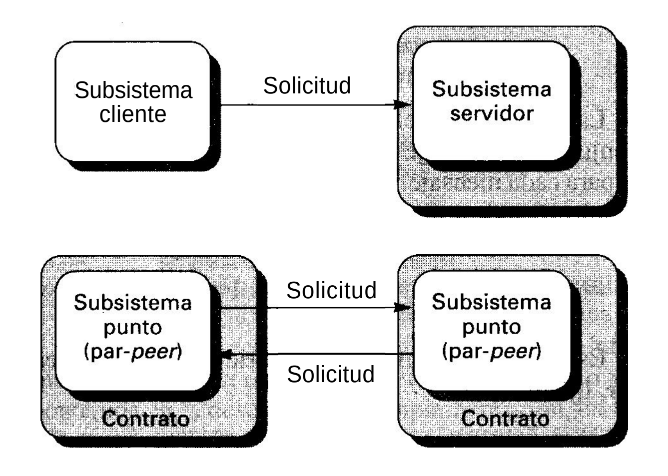


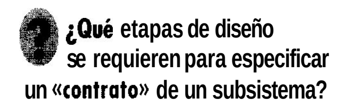
FIGURA 22.4. Modelo de colaboración entre subsistemas.

22.2.6. Componente de gestión de recursos

Están disponibles una variedad de recursos diferentes para un sistema o producto **OO;** y, muchas veces, los subsistemas compiten por estos recursos al mismo tiempo. Los recursos globales del sistema pueden ser entidades externas (por ejemplo, una unidad de disco, procesador o línea de comunicaciones) o abstracciones (por ejemplo, una base de datos, un objeto). Sin importar la naturaleza del recurso, el ingeniero de software debe diseñar un mecanismo de control para ello. Rumbaugh y sus colegas [RUM91] sugieren que cada recurso deba ser poseído por un «objeto guardián». El objeto guardián es el portero del recurso, controlando el acceso y moderando peticiones contradictorias para **él.**

22.2.7. Comunicaciones entre subsistemas

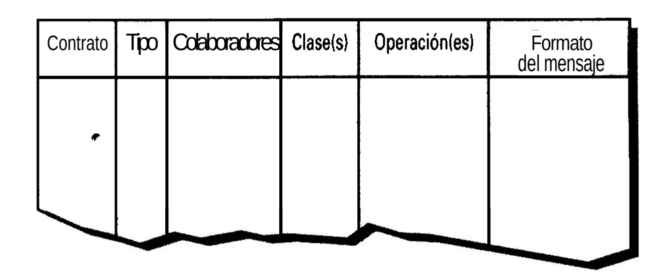
Una vez que cada subsistema ha sido especificado, es necesario definir las colaboraciones que existen entre subsistemas. El modelo que se usa para la colaboración objeto-objeto puede ser extendido en conjunto para los subsistemas. La Figura **22.4** ilustra un modelo de colaboración. Como se vio anteriormente en este capítulo, la comunicación puede ocurrir estableciendo un enlace cliente/servidor o punto-a-punto. Refiriéndose a la Figura, se debe especificar la interacción que existe entre subsistemas. Recuérdese que un contrato proporciona la indicación de los modos como un subsistema puede interactuar con otro.



Las siguientes etapas de diseño pueden aplicarse para especificar un contrato para un subsistema [WIR90]:

1. *Listar cada petición que puede ser realizada por los colaboradores del subsistema.* Organizar las peticiones por subsistema y definirlas dentro de uno o más contratos apropiados. Asegurarse de anotar contratos que se hereden de superclases.
2. *Para cada contrato, anotar las operaciones (las heredadas* y *las privadas,) que se requieren para implementar las responsabilidades que implica el contrato.* Asegurarse de asociar las operaciones con las clases específicas, que residen en el subsistema.
3. *Considerar un contrato a la vez, crear una tabla con la forma de la Figura 22.5. Para cada contrato, la tabla debe incluir las siguientes columnas:*

   Tipo: el tipo de contrato (por ejemplo, cliente/servidor o punto-a-punto).



Contrato	Tipo	Colaboradores	Clase(s)	Operación(es)	Formato del mensaje

FIGURA 22.5. Tabla de colaboraciones del subsistema.

[9] Los lectores interesados deben consultar [BRO91], [TAY92] y [RAO94]

Colaboradores: los nombres de los subsistemas que son parte del contrato.

Clase: los nombres de las clases (contenidas en el subsistema), que proporcionan servicios denotados por el contrato.

Operaciones: nombres de las operaciones (dentro de la clase), que implementan los servicios.

Formato del mensaje: el formato del mensaje requerido para implementar la interacción entre colaboradores.

Esboce una descripción apropiada del mensaje, para cada interacción entre los subsistemas.

4. *Si los modos de interacción entre los subsistemas son complejos, debe crear un diagrama subsistema-colaboración como el de la Figura 22.6.* El grafo de colaboración es similar, en forma, al diagrama de flujo de sucesos examinado en el Capítulo 21. Cada subsistema se representa, junto con **sus** interacciones con otros subsistemas. Los contratos que **se** invocan durante interacción, se detallan como se muestra en la Figura. Los detalles de la interacción se determinan observando el contrato en la tabla de colaboración del subsistema (Fig. 22.5).



Coda contrato entre subsistema se manifiesto por uno o más mensajes que se transportan entre los objetos dentro de los subsistemas.

22.3 PROCESO DE DISEÑO DE OBJETOS

Retomando la metáfora que se introdujo al inicio del libro, el diseño de sistemas OO se puede visualizar como el plano del suelo de una casa. El plano del suelo especifica **el** propósito de cada habitación y **sus** características arquitectónicas, que conectan las habitaciones unas con otras y con el ambiente exterior. Ahora es el momento de proporcionar los detalles que se requieren, para construir cada habitación. En el contexto del DOO, el diseño de objetos se centra en las «habitaciones».

Bennet **y sus** colegas [BEN99] examinan el diseño de objetos de la siguiente manera:

> **El diseño de objetos tiene que ver con el diseño detallado de los objetos y sus interacciones. Se completa dentro de la arquitectura global, definida durante el diseño del sistema y de acuerdo con las reglas y protocolos de diseño aceptados. El diseño del objeto está relacionado en particular con la especificación de los tipos de atributos, cómo funcionan las operaciones y cómo los objetos se enlazan con otros objetos.**

Es en esta fase cuando los principios y conceptos básicos asociados con el diseño a nivel de componentes (Capítulo **16**) entran en juego. Se definen las estructuras de datos locales (para atributos), y se diseñan los algoritmos (para operaciones).

22.3.1. Descripción de objetos

Una descripción del diseño de un objeto (instancia de clase o subclase) puede tomar una o dos formas [GOL83]: (1) *Una descripción de protocolo* que establece la interfaz de un objeto, definiendo cada mensaje que el objeto puede recibir y las operaciones que el objeto lleva a cabo cuando recibe un mensaje, o (2) *Una descripción de implementación* que muestra detalles de implementación para cada operación implicada por un mensaje pasado a un objeto. Los detalles de implementación incluyen información acerca de la parte privada del objeto; esto significa, detalles internos acerca de la estructura de datos, que describen los atributos del objeto, y detalles de procedimientos, que describen las operaciones.



Asegúrese de que lo arquitectura se ha definido antes de comenzar el diseño de objetos. No deje que lo arquitectura predomine.

La descripción del protocolo no es nada más que un conjunto de mensajes y un comentario correspondiente para cada mensaje. Por ejemplo, una porción del protocolo de descripción para el objeto sensor de movimiento (descrito anteriormente), podría ser:

```
MENSAJE(sensor.movimiento) —-> leer: DEVUELVE sensor.ID, sensor.estado;
```

Esto describe el mensaje requerido para leer el Sensor. Igualmente,

```
MENSAJE (sensor movimiento) —-> asigna: ENVIA sensor.ID, sensor.estado;
```

Asigna (establece) o inicializa el estado del Sensor.

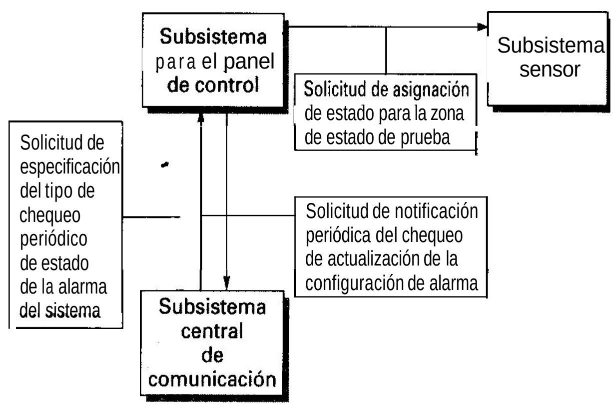


FIGURA 22.6. Gráfico abreviado del subsistema colaborado de *HogarSeguro.*

Para un sistema grande con muchos mensajes, generalmente es posible crear categorías de mensajes. Por ejemplo, categorías de mensajes para el objeto *Sistema* de *HogarSeguro* deberían incluir mensajes de configuración del sistema, mensajes de monitorización (supervisión), mensajes de sucesos, etc.

Una descripción de la implementación de un objeto, proporciona los detalles internos («ocultos»), que se requieren para la implementación, pero no son necesarios para ser invocados. Esto significa que el diseñador del objeto debe proporcionar una descripción de implementación, y debe por tanto crear los detalles internos del objeto. Sin embargo, otro diseñador o desarrollador que utilice el objeto u otras instancias del objeto, requiere solo la descripción del protocolo, pero no la descripción de la implementación.

Una descripción de la implementación se compone de la siguiente información: (1) una especificación del nombre del objeto y referencia a la clase; (2) una especificación de las estructuras de datos privadas, con indicación de los datos y sus respectivos tipos; *(3)* una descripción de procedimientos de cada operación o, alternativamente, indicadores a dichas descripciones de procedimientos. La descripción de implementación debe contener información suficiente, para el manejo adecuado de todos los mensajes descritos en la descripción de protocolo.



Para alcanzar los beneficios del ocultamiento de información (Capítulo 13), cualquiera que intente usar un objeto solo necesita la descripción del protocolo. La descripción de la implementación contiene detalles, que deberían «ocultarse» de aquellos que no tienen necesidad de conocer.

Cox [COX85] caracteriza la diferencia entre la información contenida en la descripción de protocolo y la contenida en la descripción de implementación, en términos de «usuarios» y «proveedores» de servicios. Un «usuario» del servicio provisto por un objeto debe ser familiar con el protocolo de invocación del servicio; eso significa especificar lo que se desea. El proveedor del servicio (el propio objeto), debe ocuparse del modo en que el servicio se suministrará al usuario; eso significa con detalles de implementación.

22.3.2. Diseño de algoritmos y estructuras de datos

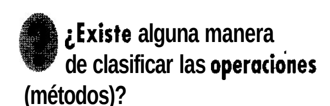
Una variedad de representaciones contenidas en el modelo de análisis y el diseño de sistema, proveen una especificación para todas las operaciones y atributos. Los algoritmos y estructuras de datos se diseñan utilizando un enfoque, que difiere un poco de los enfoques del diseño de datos y del diseño a nivel de componentes examinadas para la ingeniería del software convencional.

Referencia cruzada

Aparentemente, cada concepto que se presentó en el Capítulo 13 es también aplicable aquí. Asegúrese de estar familiarizado con los temas presentados en el Capítulo 13.

Se crea un algoritmo para implementar la especificación para cada operación. En muchas ocasiones, el algoritmo es una simple secuencia computacional o procedural, que puede ser implementada como un módulo de software autocontenido. Sin embargo, si la especificación de la operación es compleja, será necesario modularizar la operación. Las técnicas convencionales de diseño de componentes se pueden usar para resolver esta tarea.

Las estructuras de datos se diseñan al mismo tiempo que los algoritmos. Ya que las operaciones manipulan los atributos de una clase, el diseño de estructuras de datos, que reflejan mejor los atributos, tendrán un fuerte sentido en el diseño algorítmico de las operaciones correspondientes.



¿Existe alguna manera de clasificar las operaciones (métodos)?

Aunque existen muchos tipos diferentes de operaciones, normalmente se pueden dividir en tres grandes categorías: (1) operaciones que manipulan los datos de alguna manera (por ejemplo, agregando, eliminando, reformateando, seleccionando), (2) operaciones que ejecutan cálculos, y *(3)* operaciones que monitorizan (supervisan) al objeto para la ocurrencia de un suceso controlado.

Por ejemplo, la descripción del proceso *HogarSeguro* contiene los fragmentos de la oración: «al sensor se asigna un número y tipo» y «una contraseña (*password*) maestra programada para habilitar y deshabilitar el sistema». Estas dos frases indican varias cosas:

- Que una operación de asignación es importante para el objeto *Sensor.*
- Que una operación *programar* se aplicará al objeto *Sistema.*
- Que las operaciones *habilitar* y *deshabilitar* se aplican a *Sistema* (finalmente se debe definir el *estado de sistema* usando la notación adecuada en un diccionario de datos) como:

```
estado del sistema = [habilitado |
                      deshabilitado]
```



Una operación se define en gran parte de la misma manera en que se refina una función en el diseño convencional. Escriba uno descripción del proceso, haga un análisis gramatical y aísle nuevas operaciones a un nivel de abstracción más bajo.

La operación ***programar*** se asigna durante el AOO, pero específicamente durante el diseño del objeto se refinará un número de operaciones más específicas, que se requieren para configurar el sistema. Por ejemplo, después de discusiones con el ingeniero del producto, el analista, y posiblemente con el departamento de marketing (mercadotecnia), el diseñador debe elaborar la descripción del proceso original, y escribir lo siguiente para ***programar*** (subrayan operaciones potenciales —verbos—):

> ***Programar*** **habilita al usuario de *HogarSeguro* para configurar el sistema una vez que ha sido instalado. El usuario puede (1) instalar números de teléfonos; (2) definir tiempos de demora para alarmas; (3) construir una tabla de sensores que contenga cada ID de sensor, su tipo y asignación; y (4) cargar una contraseña (password) maestra.**

Por consiguiente, el diseñador ha refinado la operación simple ***programar,*** y se reemplaza con las operaciones: ***instalar, definir, construir y cargar.*** Cada una de estas nuevas operaciones se vuelve parte del objeto Sistema, que tiene conocimiento de las estructuras de datos internas, que implementan los atributos de los objetos, y que se invoca enviando mensajes con el formato:

```
MENSAJE(sistema) —> instalar: ENVÍA teléfono.número;
```

que implica que se proporciona al sistema un número de teléfono de emergencia, *y* un mensaje ***instalar se envía al*** sistema.

Los verbos denotan acciones u ocurrencias. En el contexto de la formalización del diseño del objeto, se consideran no sólo verbos, sino frases verbales descriptivas y predicados (por ejemplo, «es igual a»), como operaciones potenciales. El análisis gramatical se aplica recursivamente, hasta que cada operación ha sido refinada a su nivel más detallado.

22.4 PATRONES DE DISEÑO

Los mejores diseñadores en cualquier campo tienen una habilidad extraña para reconocer patrones que caracterizan un problema **y** los patrones correspondientes, que pueden combinarse para crear una solución. Gamma y **sus** colegas [GAM95] examinan esto cuando afirman que:

> **Se encontrarán patrones repetidos de clases y objetos de comunicación, en muchos sistemas orientados a objetos. Estos patrones resuelven problemas específicos de diseño, y vuelven al diseño orientado a objetos más flexible, elegante y extremadamente reutilizable. Ayudan a los diseñadores a reutilizar diseños exitosos basando nuevos diseños en experiencia previa. Un diseñador bastante familiarizado con ese tipo de patrones puede aplicarlos inmediatamente a problemas de diseño, sin tener que redescubrirlos.**

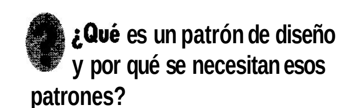
Referencia cruzada
Los patrones existen a nivel tanto de arquitectura como de componentes. Para mayor información, véase el Capítulo 14.

Durante el proceso de DOO, un ingeniero del software debe observar cada oportunidad en la que puedan reutilizar patrones de diseño existentes (cuando cumplen las necesidades del diseño), en vez de crear otros nuevos.

22.4.1. ¿Qué es un patrón de diseño?

Antes de examinar los patrones con detalle vale la pena observar un ejemplo sencillo de un patrón, que se presenta una y otra vez. Muchas aplicaciones tienen el requisito de que solo una instancia de un solo objeto debe ser instanciada. Algunos ejemplos de aplicaciones y objetos de instancias simples son:

- En un sistema operativo habrá solo un objeto administrador de archivos, que mantiene el registro de los archivos del usuario, y proporciona facilidades para crear, renombrar y eliminar tales archivos. Solo existirá una instancia de ese administrador de archivos.
- En un sistema de control de tráfico aéreo, solo existirá una instancia del controlador, que mantiene registros de los planes de vuelo y **sus** posiciones.
- En una aplicación bancaria, solo existirá un controlador, que mantiene el registro de los cajeros automáticos utilizados por el banco.


¿Qué es un patrón de diseño y por qué se necesitan esos patrones?

La Figura 22.7 muestra la estructura general de este patrón, la palabra «static» describe una variable utilizada para toda la clase. En esta Figura solo se muestran dos operaciones en la clase ***Singleton,*** pero se pueden mostrar algunas más dependiendo del contexto. A continuación se muestra un esqueleto en Java para el patrón:

```
public class Singleton {
private static ObjetoSimple instanciaúnica = null;
    public static ObjetoSimple instanciaUnica ( )
    {
        if (instanciaúnica == null )
        instanciaúnica = new ObjetoSimple ( );
        return instanciaúnica;
    }
//Código para constructores, será privado.
//Código para métodos que implementen la escritura de
//operaciones dentro de Singleton, que pueden incluir
```

```
//operacion1 y operacion2.
//Código para métodos que implementen operaciones de retorno
//en el objeto Singleton, debe incluir operación1
//operación2
}
```

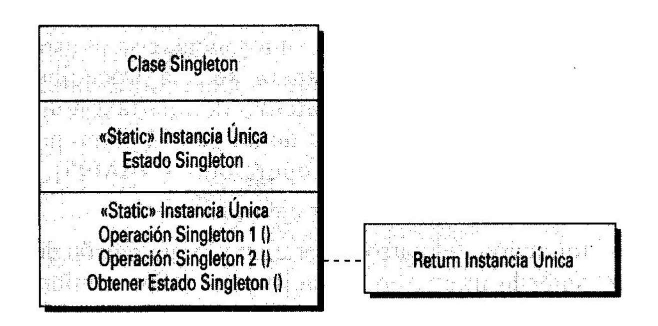


FIGURA 22.7. La estructura general del patrón *Singleton.*

El patrón ***Singleton*** se implementa por medio de una variable de instancia estática, descrita por el atributo de clase **ObjetoSimple**. La cual es inicializada con null para la instanciación.

El acceso al objeto Singleton se realiza mediante el método ***instancialrnica.*** Este comprueba primero si ObjetoSimple es igual a ***null.*** En caso afirmativo, significa entonces que no se ha creado un objeto de tipo Singleton, y que el método llamará a un constructor privado adecuado para establecer al objeto Singleton; en el código anterior esto se realiza cuando el argumento del constructor es cero. El constructor utilizado se declarará como privado, ya que no se desea **que** ningún usuario pueda crear objetos de tipo Singleton, excepto si es por medio de ***instancialrnica,*** la cual se ejecutará, de una vez y por todas, en el momento de la creación. La clase también contendrá métodos, que ejecutarán operaciones en un objeto de tipo Singleton.

De este modo, si el patrón Singleton se utilizara en un sistema de control de tráfico aéreo, y solo se necesitara un controlador de aeronaves, entonces el nombre de la clase anterior debena ser **ControladorAéreo,** y debería tener métodos tales como ***obtenerControladorAéreo***, que devuelve la Única instancia de un controlador de tráfico aéreo.

22.4.2. Otro ejemplo de un patrón

Se ha visto ya un ejemplo de un patrón: ***Singleton.*** El objetivo de esta sección es describir un patrón más complicado, conocido como ***Memento.***



El patrón de Memento se usa para la recuperación de sistemas.

El rol de Memento es el de vigilar el estado o almacenamiento de recuperación del estado de un sistema, cuando se requiera. La Figura 22.8 muestra el diagrama de clase para el patrón. Existen tres elementos de este patrón. El primero es **ClaseCreadora**, el cual es una clase que describe objetos cuyo estado debe ser almacenado. Existen dos métodos asociados con esta clase: ***fijarValorMemento*** y ***construirMemento.*** El primero inicializa su estado, toma un argumento el cual es un objeto definido por la clase Memento y reestablece su estado con el uso del argumento. El segundo crea un objeto de la clase **ClaseMemento** la cual contiene su estado actual. En efecto, ***fijarMemento*** actúa como un recurso de restauración, mientras que ***construirMemento*** lo hace como un recurso de almacenamiento.

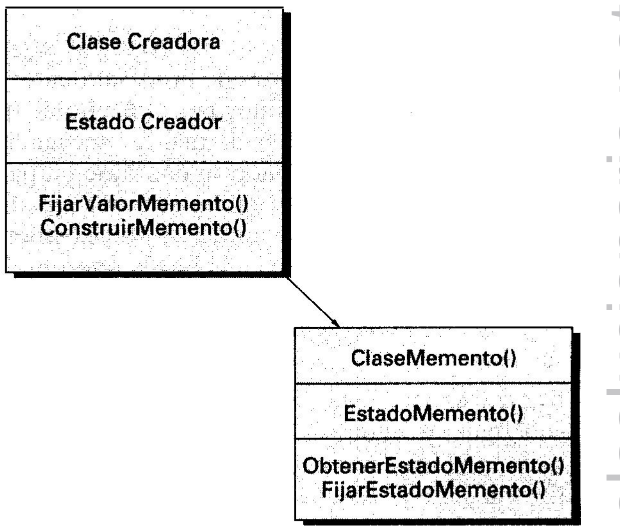


FIGURA 22.8. El patrón *Memento.*

La clase **ClaseMemento** define objetos que mantendrán el estado de un objeto de la clase ClaseCreadora. Contiene dos métodos ***obtenerEstadoMemento*** y ***fijarEstadoMemento.*** La primera devuelve el estado que se almacena, mientras que la segunda fija el estado a un valor pasado como argumento. El tercer elemento del patrón Memento es la clase cliente **Caretaker,** ésta no se muestra en la Figura 22.8. Representa una clase que implementa objetos que contienen un objeto de la clase **ClaseMemento.** Tiene un conjunto muy limitado de funciones ya que todo lo que realiza es almacenar un Memento, no altera ni examina los contenidos de un memento.

22.4.3. Un ejemplo final de un patrón

Frecuentemente, existe la necesidad de desarrollar un código que lleve a cabo el procesamiento de datos accedidos secuencialmente. Este acceso secuencia1 tendrá usualmente la misma forma, y por

esto es adecuado para un patrón. El objetivo de esta sección es describir tal patrón; ello es debido a Grand [GRA99]. Algunos ejemplos de procesamiento secuencial son:

- Un programa de informes debe procesar un archivo de datos de empleados, leyendo cada registro y desechando todos aquellos registros de empleados que ganen por encima de una cierta cantidad.
- Un programa editor puede ser consultado por su usuario, para listar las líneas de texto de un archivo que coincidan con un cierto patrón.
- Un programa de análisis de la Web debe leer el código fuente de un documento HTML, para descubrir cuantas referencias a otros sitios contiene el documento.


¿Qué hace el Filtro?

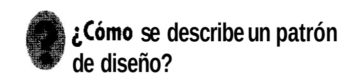
Estas son formas diferentes de procesamiento; de cualquier manera, están unidos por el hecho de que el procesamiento de datos es de manera secuencial. Este procesamiento debe hacerse con base en palabra por palabra, o con base en registro por registro; a pesar de ello, también se aplica al procesamiento secuencial, y de aquí que sea una buena oportunidad de capturarse en un patrón. Este patrón, conocido como *Filtro,* se muestra en la Figura 22.9. Consta de varias clases:

- **FiltroFuente.** Esta es la clase que actúa como superclase para otras clases concretas, que implementan el procesamiento requerido. La clase no lleva a cabo la recuperación de los datos que se procesarán, pero delega eso al objeto *Fuente,* cuya clase será descrita en la tercera viñeta siguiente. El objeto *Fuente* se pasa por medio del constuctor a la clase. La clase contiene un método llamado **obtenerDatos,** que recupera los datos que se procesarán.
- **FiltroFuenteConcreto.** Esta es una subclase de la clase FiltroFuente. Redefine el método obtenerDatos, para realizar algunas operaciones extras en los datos que han sido leídos, por ejemplo si el patrón se utiliza para contar las cadenas de caracteres que coinciden con cierto patrón, el código para realizar este recuento se coloca aquí. Normalmente este método utiliza el método correspondiente obtenerDatos, dentro de la super-clase.
- **Fuente.** Esta clase se asocia con los objetos, que llevan a cabo el proceso de recuperar los datos que se procesarán. Esto se logra por medio de un método llamado obtenerDatos.
- **FuenteConcreta** Esta clase es una subclase de Fuente e implementa el método obtenerDatos. Su función es proporcionar datos a los objetos asociados con las clases FiltroFuenteConcreto, que realizarán algún tipo de procesamiento con los datos.

22.4.4. Descripción de un patrón de diseño.

Las disciplinas maduras de ingeniería hacen un vasto uso de patrones de diseño. La ingeniería del software solo se encuentra en su infancia, con el uso de patrones. De cualquier manera, se está procediendo rápidamente hacia el comienzo de una taxonomía. En general, la descripción de un patrón como parte de una taxonomía debe proporcionar [GAM95]:

- Nombre del patrón. Por ejemplo *Filtro.*
- Intención del patrón. Por ejemplo, un patrón debe tener la intención de facilitar el mantenimiento, pues puede acomodar un número de diferentes tipos de objeto.
- Los problemas de diseño que motivan el patrón. Por ejemplo, un patrón debe desarrollarse, para que un número de transformaciones diferentes de datos puedan ser aplicadas a un objeto, muchas de las cuales son desconocidas, cuando el patrón es desarrollado originalmente.


¿Cómo se describe un patrón de diseño?

- La solución que resuelve estos problemas.
- Las clases que se requieren para implementar la solución. Por ejemplo, las cuatro clases descritas en la Figura 22.9.
- Las responsabilidades y colaboraciones entre las clases de solución. Por ejemplo, una clase debe ser responsable de la construcción de un objeto, cuyo comportamiento varía al tiempo de ejecución.
- Lineamientos que conduzcan a una implementación efectiva. Generalmente, existirá un número de formas diferentes de programar un patrón; por ejemplo, el procesamiento de errores debe ser tratado de diferentes formas.
- Ejemplos de código fuente.
- Referencias a otros patrones de diseño, o patrones que puedan usarse en conjunto con el patrón descrito.

22.4.5. El futuro de los patrones

En la actualidad, se ha desarrollado y catalogado un número relativamente pequeño de patrones. De cualquier manera, los últimos cinco años han sido testigos de una tremenda explosión, en términos de interés industrial. Los patrones, junto con los componentes, ofrecen la esperanza de que la ingeniería de software, eventualmente, se parezca a otras dis-

ciplinas de ingeniería, con clases volviéndose el análogo en software de componentes electrónicos, y los patrones volviéndose el análogo en software de pequeños circuitos hechos de componentes. Antes de que esto suceda, existe aún una ingente cantidad de trabajo que llevar a cabo al identificar patrones y catalogarlos; también, se producirá esta situación, cuando el número de patrones existentes se vuelva tan grande, que se necesite una forma de indexado automática o semiautomática.



CLAVE

En la actualidad existe un buen grupo de patrones; sin embargo, en los próximos años debería haber una verdadera expansión en su uso.

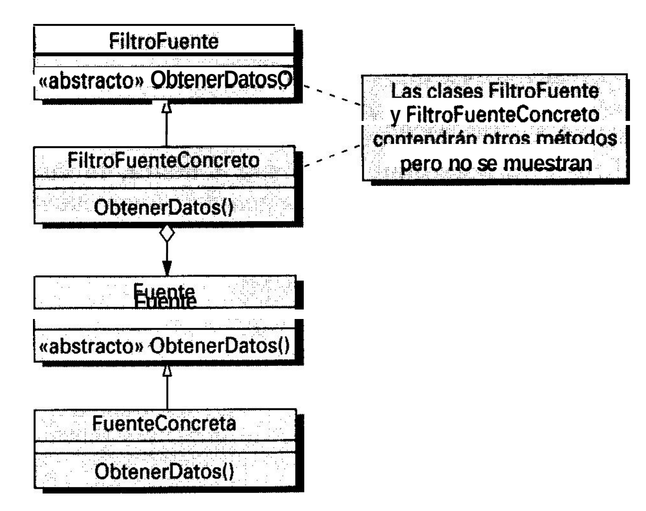


FIGURA 22.9. El patrón *Filtro.*

22.5 PROGRAMACIÓN ORIENTADA A OBJETOS

El objetivo de esta sección es describir, con un grado de mayor detalle, el conjunto de notaciones que componen el lenguaje UML. Anteriormente, en el Capítulo 21, se describen su origen y componentes principales; de hecho, muchos de los diagramas presentados en el Capítulo 21 y **en** este capítulo han sido expresados en UML. Se ha tomado la decisión de utilizar esta notación, porque se ha vuelto predominante muy rápidamente en aquellas compañías que utilizan ideas de ingeniería para desarrollar software orientado a objetos. El primer componente que se intenta describir es el modelo de clases.



CLAVE

UML se está convirtiendo en un estándar de facto para el análisis y diseño orientado a objetos.

22.5.1. El modelo de clases

Un modelo de clases es una descripción de las clases en un sistema y sus relaciones. No describe el comportamiento dinámico del sistema, por ejemplo el comportamiento de objetos individuales. El primer elemento de un diagrama de clases es una descripción de clases individuales. La Figura 22.10 muestra como se describe una clase. La clase describe al cliente de un banco.

Esta figura es muy simple, ya que solo contiene una clase. Consta del nombre de la clase **(Cliente),** el nombre de algunos de sus atributos, por ejemplo el atributo *dirección,* que contiene la dirección del cliente, y la lista de operaciones; por ejemplo, la operación *obtenerNombre* recupera el nombre de un cliente. Cada cuadro que representa una clase contiene, por consiguiente, una sección que contiene el nombre de la clase, una sección que enumera los atributos de los objetos definidos por la clase, y una sección que describe las operaciones asociadas con tales objetos.

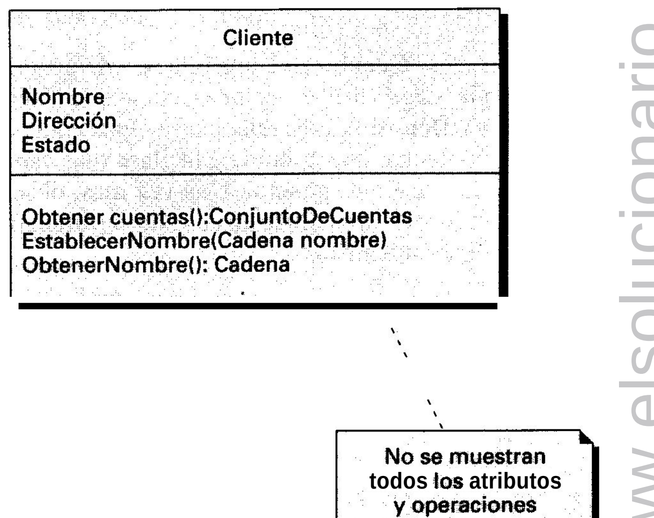


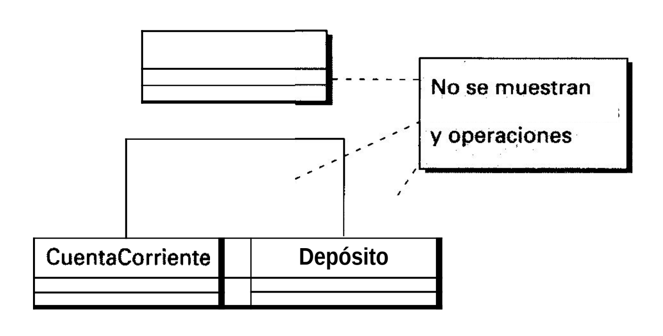
FIGURA 22.10. Un ejemplo de una clase descrita en UML.

También en la Figura 22.10 se muestra una versión gráfica de los comentarios utilizados en el lenguaje de programación. El cuadro, con una esquina superior derecha doblada, llama la atención del lector a algún aspecto de un diagrama. En el caso de la Figura 22.10, se llama la atención del lector, al hecho de que muchos de los atributos y operaciones asociados con la clase Cliente no se muestran; por ejemplo, la colección de cuentas que posee el cliente no se muestran en la sección de atributos de la clase.

La Figura 22.10 es muy simple, en la práctica los diagramas de clases en UML mostrarán la relación entre clases. Existe un gran número de tipos diferentes de relaciones, que pueden ser expresadas. La primera cosa que tratemos será la generalización.

22.5.2. Generalización

Esta relación es la que se mantiene entre una clase **X** y otra clase **Y,** cuando la clase Y es el ejemplo más específico de la clase **X.** Por ejemplo, una relación de generalización existe entre una clase **Cuenta,** la cual representa una cuenta bancaria general, y una cuenta corriente, que es un ejemplo específico de una cuenta. La Figura 22.11 muestra como se representaría esta relación en un diagrama de clases UML.



Aquí una clase Cuenta tiene una relación de generalización con las clases más específicas, como son **CuentaCorriente** y **Depósito.** Esta relación se representa por medio de una flecha, que apunta de la clase más específica hacia la clase más general. Una vez más, observe que, para propósitos ilustrativos, no se muestra ninguna operación o atributo.

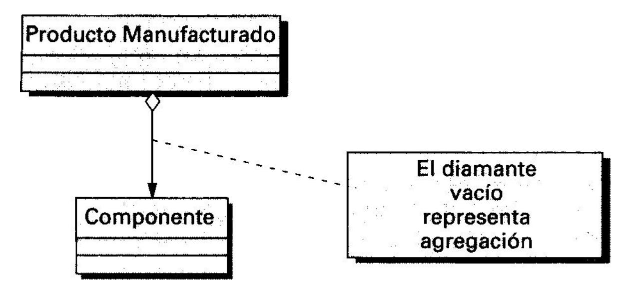


FIGURA 22.12. Agregación en UML.

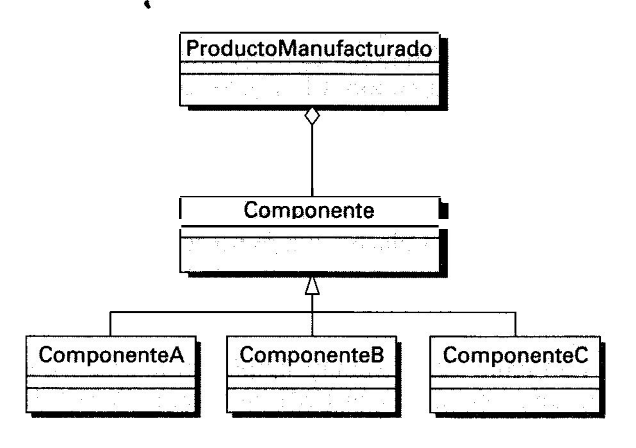
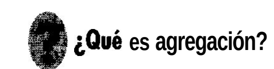


FIGURA 22.13. Un diagrama UML de clases mostrando generalización y agregación.

22.5.3. Agregación y composición

La sección anterior describió una relación, que puede ser representada en un diagrama de clases UML: generalización; las otras relaciones importantes son la agregación y la composición. Existen dos relaciones que establecen que una clase genera objetos, que son parte de un objeto definido por otra clase. Por ejemplo, un sistema para un fabricante tendrá la necesidad de mantener los datos acerca de los elementos que se están fabricando, y de aquellos que *se* están haciendo. Por ejemplo, un ordenador se fabricaría a partir de una extensa serie de componentes incluyendo su armazón, un disco duro, un conjunto de tarjetas de memoria, etc. El ordenador se construye, a partir de una serie de componentes y en un sistema orientado a objetos utilizado para dar soporte al proceso de fabricación, existirá una relación de agregación entre la clase utilizada para describir el producto fabricado y cada uno de sus componentes. La Figura 22.12 muestra como se representa esta relación, en un diagrama de clases UML.



Aquí la línea con un rombo hueco en un extremo indica que la clase describe objetos que agregan otros objetos, la clase con el rombo unido a ella describe objetos, que contiene objetos definidos por la otra clase. En UML las relaciones normalmente se mezclan. Por ejemplo, la Figura 22.12 **habrá** un número de componentes, que posean una relación de generalización con la clase Componente.

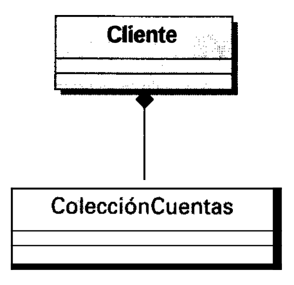


FIGURA 22.14. Un diagrama UML de clases mostrando composición.

Esto se muestra en la Figura 22.13, donde Componente se asocia con un número de clases más específicas, que describen los componentes con los que un producto fabricado puede ser ensamblado.



Existe una forma especial de agregación, conocida como composición. Ésta se usa cuando se tiene una situación en la que un objeto contiene un número de otros objetos, y cuando el objeto contenedor se elimi-

na todas las instancias de los objetos que están contenidos desaparecen. Por ejemplo, una clase Cliente que representa clientes en un banco tendrá una relación de composición con las cuentas que el cliente posee; porque si el cliente se elimina, por ejemplo, se mueve a otro banco, todas sus cuentas son eliminadas; esta forma de relación se indica de manera muy similar a la agregación, pero en esta ocasión el rombo está relleno en lugar de hueco. Esto se muestra en la Figura 22.14.

22.5.4. Asociaciones

La agregación y la composición son ejemplos específicos de una relación entre dos clases. Una relación ocurre entre dos clases cuando existe alguna conexión entre ellas, en UML esto se conoce como asociación. Algunos ejemplos de esto se describen a continuación:

- Una clase **Administrador se** relaciona con la clase **Empleado** en virtud de que un administrador dirige a un número de empleados.
- Una clase **Vuelo** se asocia con la clase **Avión** en virtud de que un avión emprende un vuelo particular.
- Una clase **Computadora** se relaciona con una clase **Mensaje** en virtud del hecho de que una colección de mensajes está esperando el proceso de una computadora.
- Una clase **InformeBancario** se relaciona con la clase **Transacción** en virtud de que el informe contiene detalles de cada transacción.

De estas relaciones, sólo la última es de agregación. Todas las otras son asociaciones claras. Tales asociaciones se escriben en UML como una línea recta. Por ejemplo, la Figura 22.15 muestra la primera asociación de las anteriores.

Las asociaciones entre clases se documentan en términos de la multiplicidad de la asociación y del nombre de la asociación. A continuación se observará la multiplicidad, examinando el ejemplo de la Figura 22.15. En este ejemplo un simple administrador dirige a uno o más empleados, y un solo empleado será dirigido por un solo administrador. Esta asociación se muestra en la Figura 22.16.

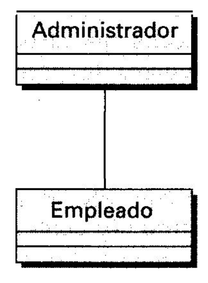


FIGURA 22.15. Un ejemplo de una relación simple en UML.

Aquí el 1 que se encuentra al final de la línea de asociación indica que un empleado solo es dirigido por un administrador; y el 1..* que se encuentra en el otro extremo de la línea determina que un administrador dirige a un conjunto de empleados.

La asociación entre clases puede nombrarse para documentar explícitamente la relación. Por ejemplo, la Figura 22.17 documenta el hecho de que un administrador dirige a un grupo (colección) de empleados.

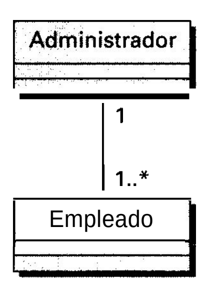


FIGURA 22.16. Multiplicidad en un diagrama de clases en UML.

Una alternativa para documentar la asociación es documentar los papeles (roles) que cada clase juega en una asociación. Un ejemplo de esta situación se muestra en la Figura 22.18. Aquí, la clase **Universidad** juega el rol de hospedar una serie de estudiantes que, a **su** vez, juegan el rol de ser estudiantes miembros de la universidad. Normalmente, cuando se documentan las asociaciones, se elige el tipo de documentación que se utilizará: si la documentación de asociación o la documentación de roles de clases que participan en la asociación. El realizar ambos, aunque perfectamente válido, se considera como un exceso.

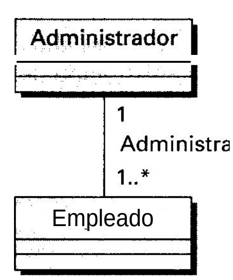


FIGURA 22.17. Documentando una Asociación.

22.5.5. Casos de uso

Anteriormente, en el Capítulo 21, se examinaron los casos de uso. En UML, un caso de uso se documenta de manera muy simple, en términos de actores y de un caso de uso. Un actor es cualquier agente que interactúa con el sistema que se construye, por ejemplo el piloto de un avión, un prestatario de libros de una librería o el jefe de los empleados en una compañía. Un caso de uso documenta alguna acción que el actor ejecuta; por ejemplo, el préstamo de un libro, el cambio de dirección de un avión o la adición de un miembro a un equipo de programación. Un caso de uso simple se muestra en la Figura 22.19. Se muestra el usuario de una biblioteca que pide prestado un libro.

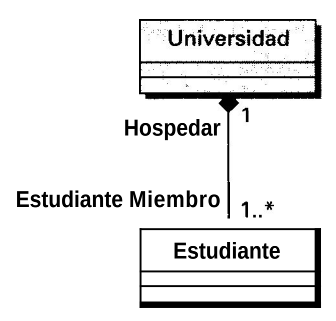


FIGURA 22.18. Documentando roles.

El actor en este caso es el prestatario, que utiliza la biblioteca, y el círculo ovalado muestra el caso de uso con el mismo nombre debajo. Los casos de uso representan una visión, a un alto nivel funcional, de un sistema en UML. En general, un sistema grande debe contener centenares, si no millares de casos de uso. Un fragmento de un grupo de casos de uso relacionados con el mismo actor se muestra en la Figura 22.20.

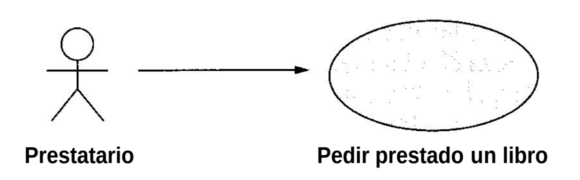


FIGURA 22.19. Un caso de uso sencillo.

Muestra algunas de las acciones que un administrador de proyecto debe llevar a cabo, cuando interactúa con un sistema de administración de proyectos.

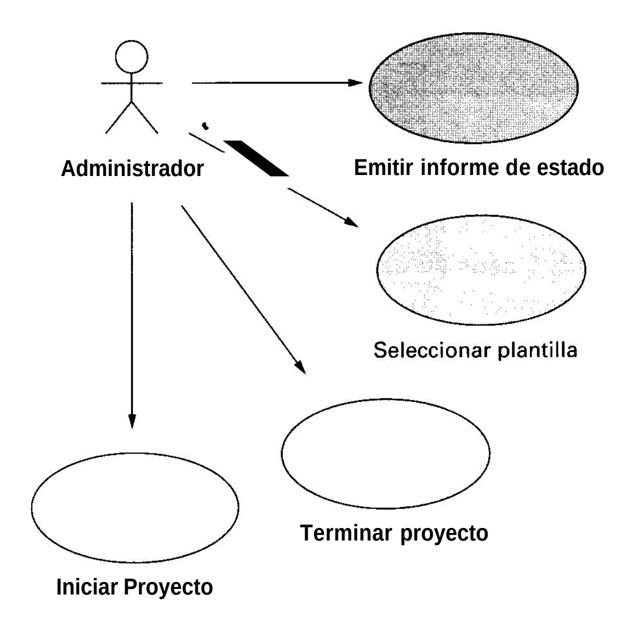


FIGURA 22.20. Un conjunto de casos de uso.

La existencia de un gran número de casos de uso significa que habrá algunos casos de uso que serán utilizados por otros casos de uso. Cuando esto sucede, el diagrama de casos de uso UML va a incluir una etiqueta conocida como un estereotipo <<uses>>, sobre la **flecha** que conduce al caso de uso. Se muestra un ejemplo en la Figura 22.21, la cual muestra algunos casos de uso, involucrados con un sistema para la administración de productos en un almacén (*Warehouse*). Por ejemplo, el administrador del almacén puede hacer una petición a nivel de existencias de un producto en particular. Al llevar a cabo estas funciones, el administrador del almacén genera un número de casos de uso, cada uno de los cuales hacen uso de otro caso de uso, que valida el nombre del producto al que se hace referencia en el caso de uso para revisar; por ejemplo, que el administrador haya escrito un nombre de producto válido.

Aquí, el hecho de que un caso de uso se emplee por otros casos, se indica por medio de una flecha con punta hueca.

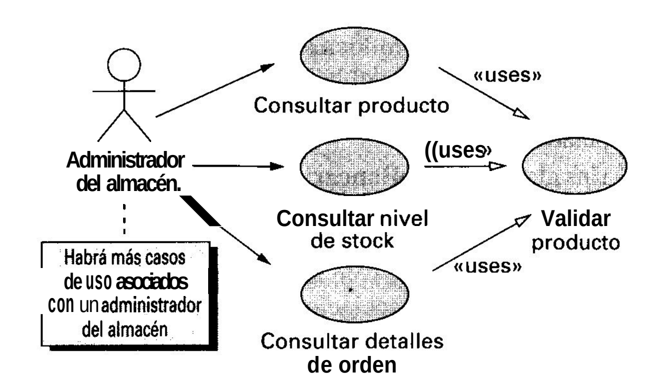


FIGURA 22.21. Un ejemplo de casos de uso utilizando otro caso de uso.

22.5.6. Colaboraciones

Durante la ejecución de un sistema orientado a objetos, los objetos interactuarán con cada uno de los otros. Por ejemplo, en un sistema bancario, un objeto **Cuenta** puede enviar un mensaje a un objeto transacción para crear una transacción que ha ocurrido en una cuenta, por ejemplo una cuenta de cargo. Este tipo de información es importante para el diseñador de un sistema orientado a objetos, durante el proceso de la identificación y validación de clases. Por esta razón, UML tiene dos notaciones equivalentes para definir las interacciones. En este libro nos centraremos sólo en uno: el diagrama de secuencias; el otro diagrama se conoce como diagrama de colaboración, y es equivalente al diagrama de secuencia; de hecho, son tan similares que las herramientas CASE pueden normalmente crear un diagrama, a partir de una instancia o de la otra. La Figura 22.22 muestra un ejemplo simple de un diagrama de secuencias.

En este diagrama existen tres objetos, los cuales se involucran en una interacción. El primero es el objeto *administrador* descrito por la clase **Empleado.** Esto envía un mensaje *actualizarInforme* a un objeto llamado *informeVentas,* el cual envía un mensaje *crearTransacción* a un objeto *transacciónVentas.*

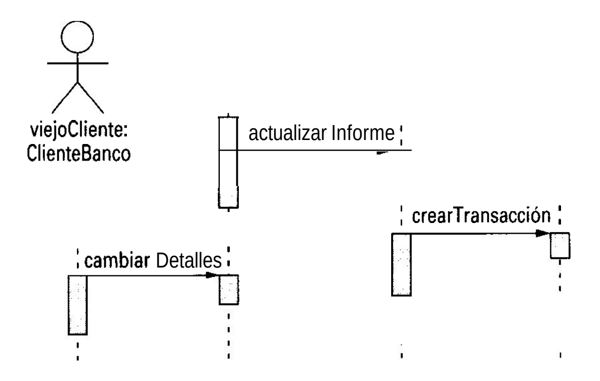


FIGURA 22.22. Un diagrama de secuencia sencillo.

En el diagrama de secuencia existen tres objetos involucrados, uno de ellos *(administrador)* tiene su clase específica **(Empleado),** los otros no. Los contenidos en los cuadros de un diagrama de secuencia pueden contener solo el nombre del objeto, el nombre de un objeto junto con su clase separado por dos puntos, o solo el nombre de una clase precedida de dos puntos; en este último caso, el objeto es anónimo.

La Figura 22.22 también muestra el rol de un actor dentro de una colaboración: aquí el actor ClienteBanco y **ViejoCliente,** interactúa con el administrador del objeto **Empleado,** enviando un mensaje *cambiarDetalles.*

La Figura 22.23 muestra otro ejemplo de un diagrama de secuencias.

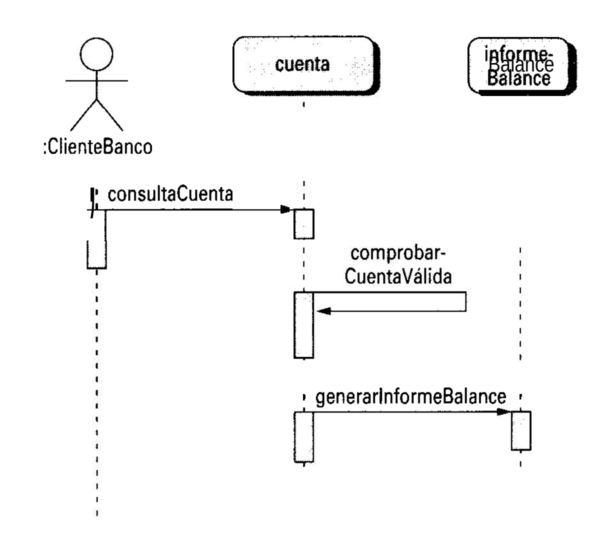


FIGURA 22.23. Otro ejemplo de diagrama de secuencia.

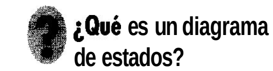
Aquí un actor, representado por un objeto anónimo definido por la clase **InformeBalance,** envía un mensaje al objeto *cuenta,* que consulta la cuenta.

Este objeto comprueba si es una cuenta válida, y luego envía un mensaje *generarInformeBalance* a un objeto *informeBalance*, que contiene los datos requeridos por el cliente del banco.

22.5.7. Diagramas de estado

Otro componente importante de UML es el diagrama de estado. Este muestra los diferentes estados en que un objeto se encuentra, y cómo se dan las transiciones de cada estado a otros estados. Tal diagrama contiene los siguientes componentes:

- *Estados:* se muestran dentro de cuadros, con esquinas redondeadas.
- *Transiciones* entre estados mediante flechas.



- *Sucesos* que provocan las transiciones entre estados.
- *Marca de inicio*, que muestra el estado inicial, en el que un objeto se encuentra cuando se crea.
- *Marca de parada (fin),* que indica que un objeto ha alcanzado el final de su existencia (vida).

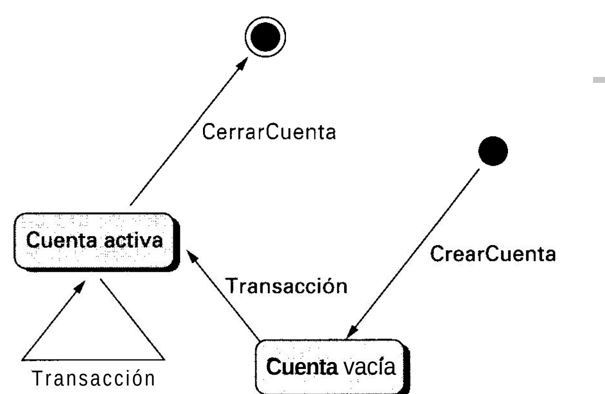


FIGURA 22.24. Ejemplo de un diagrama de estados.

Un ejemplo de diagrama de estados se muestra en la Figura 22.24.

Aquí se muestra el ciclo de vida de una cuenta bancaria. Cuando la cuenta se crea, se visualiza como una cuenta vacía. Hasta que una transacción se lleve a cabo (los fondos depositados o retirados de la cuenta se visualizan como una cuenta activa). El diagrama de estado también muestra que, cuando una cuenta se cierra, se destruye.

22.6 CASO DE ESTUDIO. LIBROS EN LÍNEA

El objetivo de esta sección es mostrar el uso de diagramas UML, descrito en la Sección 22.25, aplicado a un sistema real. Este sistema es un sistema de comercio electrónico (e-commerce).

22.6.1. Libros-en-línea

Libros-en-línea es una compañía creada recientemente, que es subsidiaria de otra gran compañía multinacional de comercio, conocida como Pollday Publishing. Los directores de Pollday Publishing se decidieron a llevar a cabo un gran crecimiento en ventas por internet entre sus clientes, y decidieron preparar a Libros-en-Línea para ello. El concepto que Pollday tiene es el de un sitio Web de comercio electrónico, que tenga catálogos detallados de cada libro que manejan, junto con recursos con los que el usuario del sitio Web puede ordenar libros, utilizando una forma incluida en una página Web. **A** continuación, se muestran algunos extractos, tomados del conjunto de requisitos iniciales, detallados por el equipo técnico de Libros-en-línea:

1. Libros-en-línea desea desarrollar la capacidad de ventas en línea por medio de la Web. EL sitio web que implementa esta capacidad debe permitir al cliente examinar los detalles del libro, ordenarlo y registrar su dirección de correo electrónico para recibir ofertas especiales con detalles, publicaciones nuevas y revisiones.
2. Cuando un cliente accede al sitio Web, cada uno de los recursos antes descritos se despliegan.
3. Si el cliente registra su dirección de correo electrónico, se le preguntarán su información personal. Esto incluye su nombre, una dirección de correo electrónico, una dirección postal. Un miembro del equipo conocido como el administrador de contratos será el responsable de enviar información de oferta, etc., a los clientes.
4. El cliente podrá comprar libros del catálogo en línea, examinando las páginas con detalles de los libros y seleccionando el libro, que comprará mediante algún mecanismo como el de presionar un botón. El sistema deberá mantener un carrito de compras virtual, en el que los detalles de cada libro se almacenan. Conforme el carrito se va llenando de libros, se muestra al cliente el precio acumulado de todas sus compras.
5. Cuando el cliente ha terminado de comprar en el sitio Web, se le pedirá información acerca de qué forma de envío se utilizará. Por omisión se mostrará la opción de envío por correo estándar, y un servicio de envío expreso garantizado para entregar dentro de 24 horas.
6. El sitio web deberá interactuar con un sistema de control de inventario, que también requiere desarrollo. Este sistema de control de inventarios debe manejar el proceso de recepción de libros de los inventarios de las editoriales, retiro de libros cuando se ordenan por parte de un cliente, y reordenación de existencia de un libro, cuando se encuentra por vaciarse, para encarar un problema de suministros, en un tiempo de siete días.
7. El control de inventarios por parte del administrador será fijado en un tiempo de siete semanas. Él o ella tendrán la responsabilidad de mantener las ventas, y la disponibilidad de existencias y, cuando las existencias de un libro se encuentren bajas, hacer un nuevo pedido. Para realizar esto, este sistema de control de inventarios debe proporcionar informes de ventas y existencias de inventario con regularidad.
8. Un Administrador de Marketting intervendrá cada ocho semanas. El Jefe de Marketting tendrá la función de determinar los precios a los que los libros serán vendidos. Se dará la situación de que un libro puede tener un número diferente de precios durante su tiempo de vida; por ejemplo, se debe decidir si se ofrece con un mayor descuento durante las primeras semanas, y luego ajustar el precio a los precios recomendados por las editoriales.

La compañía que desarrolló el software para Libros-en-línea, primero identificó un número de clases candidatas, que a continuación se detallan:

- **RegistroCliente.** Detalles de alguien que compra libros, o que se registró para recibir correos electrónicos con información.
- **Libro.** El artículo principal, qué Libros-en-línea vende.
- **Orden.** Una orden que un cliente realiza, para uno o más libros. Esta orden debe ser para una simple copia de un libro, o para copias de un número de libros o incluso múltiples copias de muchos libros. Una orden contendrá un número de líneas de orden (véase a continuación), y una especificación de envío.
- **LíneaDeOrden.** Esta es una simple línea de orden. Por ejemplo la orden de la copia de un libro. Una orden consistirá en una o más líneas de orden. Una línea de orden contendrá información acerca de qué libro se ordena y la cantidad ordenada (usualmente 1).
- **Carrito. Es** un contenedor que existe durante la exploración del sitio Web, por parte de un cliente que realiza una orden. Los contenidos del carrito serán líneas de orden. Cuando un cliente complete una orden, las líneas de orden del carrito y la información de envío proporcionada por el usuario serán anexadas a una orden.

- **OrdenEspera.** Esta es una parte de la orden, la cual no puede cumplirse por las existencias del almacén. Si el cliente está conforme, esperando por un libro que no está en existencia, entonces se realiza una orden de espera. Esta orden de espera se satisface, cuando las existencias del libro se restauran por Libros-en-línea.
- **Almacén.** Esta es una colección de libros que se encuentran en existencia. Una orden de libro o de colección de libros se manda al almacén y de ahí se retiran los libros de sus cajas del almacén, se empaquetan y se despachan al cliente. En ese momento, se ajustan los detalles de las existencias.
- **RegistroExistencias.** Estos son los datos que describen los detalles de las existencias de un libro; por ejemplo, cuántos hay en existencia, el nivel actual cuando *se* ha hecho una requisición a las editoriales, y la localización de los libros dentro del almacén.
- **NotaEmpaque.** Esta es una nota enviada con una colección de libros al cliente. La nota de empaque contiene información acerca de cuántos libros se enviaron y la tarifa de envío aplicada. También puede contener detalles de algunos libros, que no pudieron ser enviados porque no se encontraban en las existencias.
- **TarjetaCrédito.** Un cliente pagará por sus libros mediante una tarjeta de crédito. El sistema permite al cliente pre-registrar su o sus tarjetas, para que no tenga que reescribirlas cada vez que haga una orden.

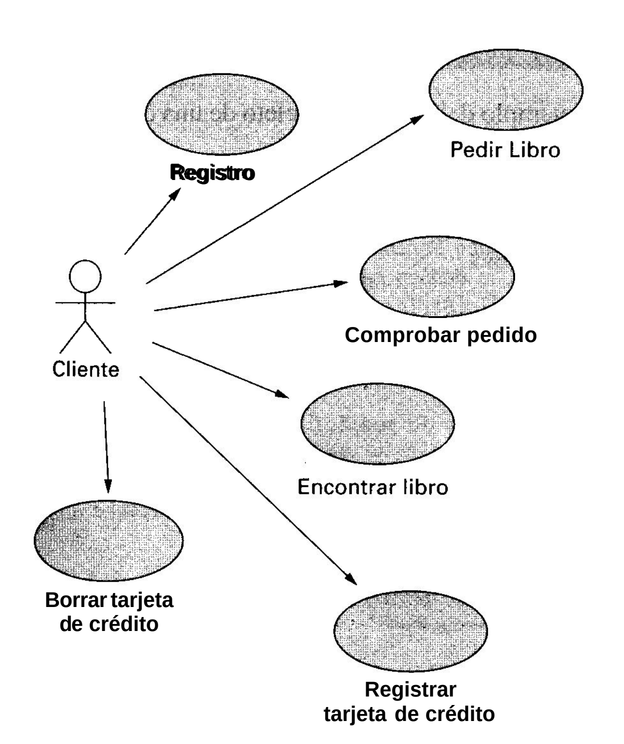


FIGURA 22.25. Algunos casos de uso para el actor Cliente.

Estas fueron, entonces, las clases identificadas principalmente. También se identificaron un número de actores:

- *Cliente.* Este es el actor principal: la persona que lleva a cabo las acciones, que resultan en los mayores cambios de estado del sistema.
- *Administrador de marketting.* Es un actor importante que ajusta muchos de los parámetros del sistema, tales como el precio de los libros.
- *Administrador del control de inventarios.* Alguien que controla los inventarios en un almacén y toma decisiones acerca de las órdenes.

Existen un gran número de casos de uso asociados con estas acciones, muchas de ellas asociados con el actor cliente, se muestran en la Figura 22.25.

La selección de casos de uso asociados con el Cliente, y las mostradas en la Figura 22.25 incluyen casos de uso para:

- *Registro.* Aquí el cliente proporciona su nombre y su clave. Una vez registrada, pueden examinar el catálogo de libros.
- *Ordenar.* Aquí el cliente ordena una o más copias de un libro.
- *Realizar.* El cliente completa la orden y ordena al sistema iniciar el proceso en que la orden se despacha.
- *Buscar un libro.* El cliente busca, en el catálogo en línea, un libro específico.
- *Eliminar tarjeta de crédito.* Aquí el cliente puede eliminar una o más de las tarjetas de crédito registradas y asociadas con él.
- *Registrar tarjeta de crédito.* Aquí el cliente registra una o más de sus tarjetas de crédito al sistema.

Una porción de uno de estos diagramas de clase para el sistema, se muestra en la Figura 22.26. Un número de roles asociados con el diagrama se ha omitido; regularmente se incluyen. Algunas de las relaciones entre clase, también se omitieron.

El diagrama muestra muchas de estas clases antes descritas. Las únicas clases que se omitieron son: **OrdenSatisfecha** y **OrdenEspera.** Estas dos clases son especializaciones de la clase **Orden,** que representa una orden de libros para un cliente.

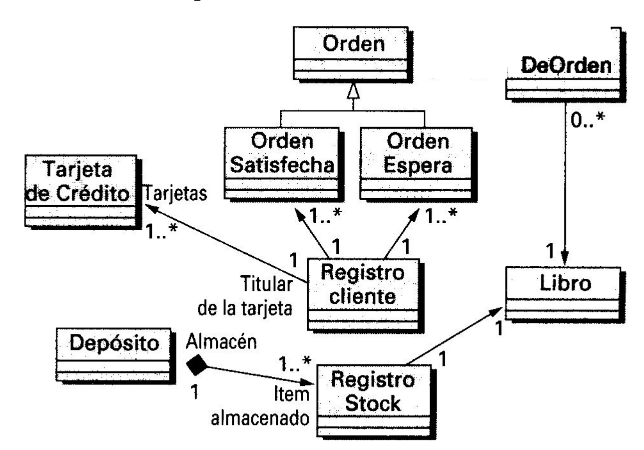


FIGURA 22.26. Una sección de un diagrama de clases para el caso de estudio.

Cuando se hace un pedido, algunos de los artículos pedidos pudieron no haberse servido, porque los libros no estaban en existencia. Cuando esto ocurre, la orden se divide en dos: todos los libros que pueden proporcionarse en un objeto OrdenSatisfecha, y aquellos que no pudieron encontrarse, se registran en un objeto OrdenEspera. Las relaciones en el diagrama de clases se detallan a continuación.

- Un almacén se asocia con un número de registros de existencias, los cuales detallan los libros almacenados en el almacén. Un simple almacén se asocia con uno o más registros de existencia.
- Un registro de existencia se asocia con un solo libro, y un libro se asocia con un solo registro de existencia.
- Una orden puede consistir en un número de líneas de orden, y una línea de orden será asociada con una sola orden.
- Un cliente registrará su número de tarjeta de crédito al sistema, un número de tarjeta de crédito se asocia con un solo cliente.
- Un cliente se asocia con un número de órdenes satisfechas, las cuales se realizan sobre un período de tiempo. Cada orden satisfecha se asocia con un solo cliente.
- Un cliente se asocia con un número de órdenes en espera, que actualmente no pueden ser satisfechas. Cada orden en espera se asocia con un solo cliente.

La Figura 22.26 muestra solo algunas de las relacienes involucradas, por ejemplo, existe una relación entre tarjetas de crédito y órdenes, en virtud de que una tarjeta de crédito particular se utiliza para pagar una orden. De cualquier manera, se muestra suficiente detalle para proporcionar una indicación de qué tan complicado se ve un diagrama de clases UML.

Un ejemplo de diagrama de secuencias asociado con el caso en estudio se muestra en la Figura 22.27.

Aquí el cliente ordena un libro. Esto resulta en el registro de existencias para el libro consultado, y se ajusta si el libro está en existencia. Si el libro está en existencia, un objeto de tipo línea de orden se crea, el cual se anexa a una orden, la cual se construye conforme el cliente navega por el sitio web de Libros-en-línea. El diagrama final (Fig. 22.28) muestra un diagrama de estado para el objeto Orden.

Un cliente primero realiza la orden, y el estado del objeto Orden se vuelve orden parcial; entonces se da al cliente la opción de añadir más libros o de eliminar libros de su orden. En cualquier momento de la construcción de la orden, el cliente puede cancelar la orden, esto conduce a la terminación. Cuando el cliente indica que se ha llegado al fin de la orden, entonces la orden se vuelve una orden de libros completa. En este punto, el cliente tiene dos opciones: cancelar al orden o especificar el tipo de envío que se usará para la orden. Si se selecciona el tipo de envío, entonces la orden se convierte en una orden completa. En esta etapa, el cliente tiene otras dos opciones: confirmar la orden, en este caso la orden se envía para ser procesada, o cancelar la orden. Ambas opciones conducen al punto de salida del diagrama de estados.

22.7 PROGRAMACIÓN ORIENTADA A OBJETOS

La etapa final de desarrollo, dentro del ciclo de vida orientada a objetos, es la programación. No es la intención de este libro llegar a más detalle acerca de este proceso; la programación es analizada como importante pero como actividad subsidiaria al análisis y diseño; pero se proporciona una pequeña introducción en el lenguaje de programación Java. La sección *otras lecturas* y *fuentes de información* al final del capítulo detalla un gran número de buenos libros sobre el tema.

El proceso de programación involucra la conversión de un diseño orientado a objetos en un código de programa. Efectivamente, esto significa que las clases definidas en el diseño deben ser convertidas en clases expresadas en un lenguaje de programación orientado a objetos como Java, C++ o SmallTalk. Esta sección se concentra en Java, que se está volviendo rápidamente el lenguaje de programación de facto, para desarrollar los modernos sistemas distribuidos.

Una clase en Java se introduce por medio de la palabra clave *class*, dentro del código para la clase; el programador define los atributos y operaciones para la clase. Un ejemplo del código esqueleto de una clase se muestra a continuación.

```
class Cliente {
    private String NombreCliente;
    private String DireccionCliente;
    // se definen más atributos aquí
    public String obtenerNombreCliente( )
    {
    //código para obtenerNombreCliente
    }
    public void modificarDireccionCliente( String
    direccion)
    {
    //código para modificarDireccionCliente
    }

    //código para las operaciones restantes
}
```

La primera línea de código define que el nombre de la clase será *Cliente*. Inmediatamente a continuación, las descripciones de atributos de clase. En el código anterior, solo se muestran dos atributos: el nombre del

cliente y **su** dirección; ambos se expresan como cadenas de caracteres. Normalmente, en un sistema real existen muchos más atributos asociados con la clase. La descripción del atributo incluye **su** tipo *(String)* y **su** visibilidad. En el ejemplo anterior, a los dos atributos mostrados se les asigna la visibilidad de privados. Esto significa que pueden ser accedidos por cualquier código dentro de la clase pero no por alguno fuera de ella; por ejemplo, el código que pertenece a otra clase. Esto significa que las variables de instancia se encapsulan dentro de la clase. Existen otros modificadores de visibilidad en Java: Se encontrará con otro después, cuando se describan las operaciones de una clase.

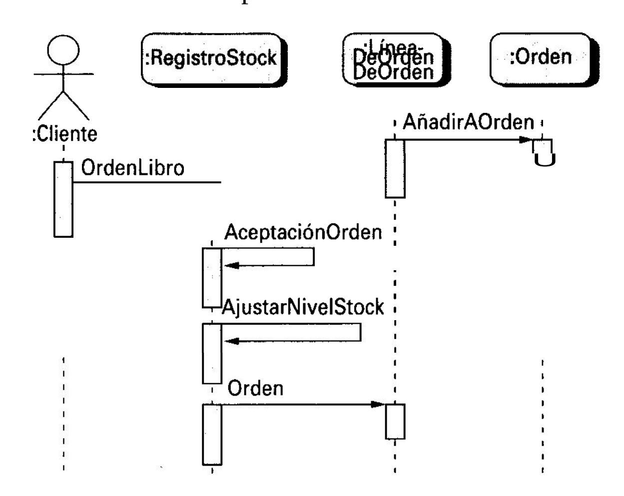


FIGURA 22.27. Un diagrama de secuencia para el caso de estudio.

También se han mostrado dos operaciones dentro de la clase Cliente. La primera es la operación *obtenerNombreCliente,* que devuelve el nombre del cliente descrito por la clase.

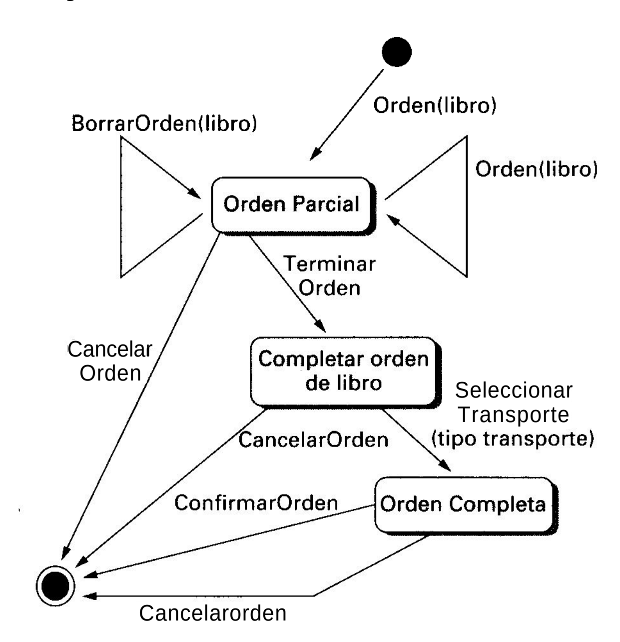


FIGURA 22.28. Un diagrama de estados.

La palabra clave *String* especifica que esta operación devolverá una cadena, y la palabra clave *public* describe el hecho de que cualquier código de cualquier otra clase puede hacer uso de esta operación: *public* es el opuesto de la palabra clave *private,* detallada en el párrafo anterior.

El código para esta operación se encierra con llaves ({}) de operación.

La operación *modificarDirecciónCliente* difiere en dos cosas de la operación *obtenerNombreCliente.* En primer lugar, le antecede la palabra *void;* esto indica que no hay resultado que regresar de la operación: la operación solo lleva a cabo acciones que modifican los atributos de la clase. En segundo, la operación asociada con el argumento *dirección,* que es una cadena que representa la nueva dirección de un cliente, que reemplazará a la anterior.

Esto, entonces, es la forma básica de una clase en Java; es muy similar en muchas formas a la estructura de clases expresada para los lenguajes orientados a objetos, con excepción de SmallTalk. **A** continuación, se muestra el código completo de una clase muy simple. La clase representa un contador, que es un dispositivo que sirve para registrar el número de veces que una página web ha sido accedida por visitantes de un sitio Web.

```
class Contador {
    private int veces;
    public Contador ( int valorInicio )
    {
        veces = valorInicio;
    }
    public void ajustaContador( int valor )
    {
        veces = valor;
    }
    public int obtenerCuenta( )
    í
        return veces;
    }
    public void incrementaCuenta( )
    {
        vecestt;
    }
}
```

La clase denominada *Contador* tiene un atributo *veces,* que registra el número de veces consultado por usuarios. La siguiente operación tiene el mismo nombre de la clase, y se conoce como constructor. Este es un fragmento de código ejecutable, que se ejecuta cuando el objeto se crea, recibe un argumento entero que es el valor inicial del contador. Cuando el usuario de la clase desea crear un objeto contador, el código es el siguiente:

```
Contador cont = new Contador( 0 );
```

La palabra clave *new* llama al constructor para crear un objeto contador que tiene un solo atributo *veces* inicializado a cero.

La operación *obtenerCuenta* regresa el valor actual del contador, la operación *ajustaContador* ajusta el atributo *veces* a un valor dado por su argumento, y la operación *incrementaCuenta* incrementa el atributo veces en uno (la operación ++ es la operación de incremento en uno).

La sintaxis de Java para enviar mensajes al objeto es la siguiente:

```
Objeto.nombreOperación( argumentos )
```

Por ejemplo para incrementar un objeto Contador cont el código sería:

```
cont.incrementaCuenta( );
```

La clase anterior es muy simple; de cualquier modo, sirve para ilustrar las características estructurales principales de cómo se define una clase. Existen otras complicaciones, como el hecho de que pueden definirse varios niveles de visibilidad; de cualquier modo, esto queda fuera del alcance de un libro de ingeniería del software.

Existen dos maneras de combinar clases: la primera es la herencia y la segunda es la agregación. Los lenguajes de programación orientada a objetos contienen recursos que permiten a ambas ser implementadas fácilmente.

En Java, la palabra clave *extends* se utiliza para derivar una clase de una ya existente por medio de la herencia. Por ejemplo, asúmase que se necesita derivar una clase nueva, que se parece mucho a la clase Contador, pero que además registra la hora a la que la cuenta se incrementó. El esqueleto de la nueva clase, que hereda la clase Contador, se muestra a continuación:

```
class TiempoContador extends Contador{
// Algunos atributos nuevos
// Algunas operaciones nuevas.
}
```

La palabra clave *extends* especifica el hecho de que la clase *TiempoContador* se hereda de la clase Contador. El código para la clase se muestra a continuación:

```
class TiempoContador extends Contador{
    Time horaAcceso;
    public TiempoContador( int valorInicio )
    {
        super( valorInicio );
        horaAcceso = new Time horaAcceso( );
    }
    public void ajustaContador( int valor )
    {
        super.ajustaContador( int valor );
        horaAcceso.setNow( );
    }
    public Time obtenHora( )
    {
        return horaAcceso;
    }
    public void incrementaCuenta( )
    {
        super.incrementaCuenta í );
        horaAcceso.setNow( );
    }
}
```

Recuérdese que, en la herencia, las operaciones en la superclase (en nuestro clase Contador) se heredan por la superclase (en nuestro caso *TiempoContador*), a menos que se sobrecarguen.

La primera operación *ajustaContador,* sobrecarga al método correspondiente en la superclase. Primero inicializa el atributo veces dentro de la superclase, haciendo una llamada a su constructor (la palabra clave súper se utiliza para ello), y luego se construye un nuevo objeto Time. Este objeto se define en cualquier otro lugar, y no se debe mostrar el código. La operación *ajustaContador* también sobrecarga la operación correspondiente dentro de Contador. El código dentro de la clase primero inicializa el atributo de la superclase, para que sea igual a valor. Luego envía un mensaje setNow al atributo *horaAcceso*, para ajustar su valor a la hora actual, el código para la operación *setNow* forma parte de la clase Time y no se muestra. El método *obtenHora* es simple: todo lo que hace es regresar el valor de *horaAcceso*. La operación *incrementaCuenta* sobrecarga la operación correspondiente en la clase Contador. Primero incrementa el valor de veces, llamando a la operación incrementaCuenta dentro de la superclase, luego ajústale valor de *horaAcceso*, enviando un mensaje *setNow* al objeto. El método *obtenCuenta,* heredado de la clase Contador, no necesita ser sobrecargada, ya que todo lo que hace es regresar el valor del atributo veces, dentro de la superclase.

Esto es, como un lenguaje de programación orientado a objetos implementa la herencia. La implementación de la agregación es más simple: todo lo que involucra es la inclusión de las partes agregadas como atributos de clase. Por ejemplo, la clase siguiente Computadora, representa a una computadora; forma parte de un sistema para planificar la fabricación de computadoras. Una computadora es agregada desde un monitor, un teclado, una unidad de proceso y así sucesivamente. Esto se muestra en la clase como una serie de atributos.

```
class Computer {
    private Monitor mon;
    private KeyBoard kb;
    private Processor proc;
    // demás atributos
    //definición de las operaciones asociadas con
la computadora.
}
```

Como un ejemplo final de código en Java, se ha reproducido mucho del código para un cliente, del pequeño caso de estudio mostrado en la Sección 22.6. Un cliente es alguien que comprará libros usando internet. El código para muchos de los métodos se encuentra a continuación:

```
class Cliente
{
    Ctring nombre, dirección, tipoTarjetaCredito,
    clave, infoSeguridad;
    HistorialCompras Hc;
    OrdenesActuales ordenesA;
    Preferencias pref;
```

```
Public obtenPrefCliente( )
{
     return pref;
}
public Ctring obtenerNombre( )
{
     return nombre;
}
public Ctring obtenDireccion( )
{
     return direccion;
}
public Ctring obtenTipoTajetaCredito( )
{
     return tipoTarjetaCredito;
}
public Ctring obtenClave( )
{
     return clave;
}
public Ctring obtenInfoSeguridad( )
{
     return infoSeguridad;
}
public void ponNombre( Ctring nom )
{
     nombre = nom;
}
public void ponDireccion( Ctring dir )
{
     direccion = dir;
}
public void ponTipoTarjetaCredito( Ctring ttc )
{
     tipoTarjetaCredito = ttc;
}
public ponClave( Ctring clv )
{
     clave = clv;
1
public void ponInfoSeguridad( Ctring isg )
{
     infoSeguridad = isg;
}
public void ponPreferencias( Ctring prf )
{
     pref = prf;
1
public void iniciaHistCompras( )
{
     //inicializa el historial de compras con
     el método setClear
     Hc.setClear( );
}
public void agregaTransCompra(TransCompra tc )
{
     //utiliza el método add definido en Histo-
     rialCompras para
     //agregar un libro a la transacción de
     compras al objeto
     //Cliente
     Hc.add( tc );
}
public HistCompras obtenHistCompras( )
{
     return Hc;
}
public void inicOrdenesA( )
{
     //utiliza el método setClear de Ordenes-
     Actuales para
     //inicializar las ordenes actuales
     ordenesA.setClear( );
}
public void agregaOrden( Orden ord )
{
     //Utiliza el método addCurrentOrders de
     OrdenesActuales
     ordenesA.addCurrentOrders( ord );
}
public void borraOrden( Orden ord )
{
     //Utiliza el método removeCurrentOrder de
     OrdenesActuales
     OrdenesA.removeCurrentOrder( ord );
}
public int numOrdenesActuales( )
{
     //Utiliza el método no de la clase Orde-
     nesActuales
     return OrdenesA.no( );
}
public OrdenesActuales obtenOrdenesActuales( )
{
     return OrdenesA;
}
...
}
```

Muchos de los métodos son muy simples: todo lo que hacen es o ajustar u obtener los valores de los atributos que se encuentran en la clase Cliente. Estos atributos son:

- *nombre.* Nombre del cliente.
- *dirección.* Dirección del cliente.
- *tipoTarjetaCrédito.* Una cadena que describe el tipo de tarjeta de crédito usada por el cliente.
- *clave.* La cadena usada por el cliente, para acceder al sitio de venta de libros.
- *infoSeguridad.* Esta es una cadena que se utiliza por el cliente, para identificarse con el equipo de servicio del sitio, en caso de que se olvide su clave. Por ejemplo, puede contener el lugar de nacimiento del cliente.
- *Hc.* Es el historial de los libros que el cliente ha comprado.
- *ordenesA.* Contiene los detalles de cada orden, que se lleva a cabo en ese momento; por ejemplo, una orden que no puede ser satisfecha completamente.
- *pref.* Contiene una lista de preferencias de compra para el cliente. Por ejemplo, el hecho de que el cliente normalmente compra novelas de terror.

Existe un grupo de métodos, que pueden llamar a métodos definidos en otras clases; por ejemplo, el método *borraOrden,* que elimina una orden de los detalles de cliente, y utiliza el método *removeCurrentOrder* de la clase **OrdenesActuales.**

RESUMEN

El diseño orientado a objetos traduce el modelo de AOO del mundo real, a un modelo de implementación específica, que puede realizarse en software. El proceso de DOO puede describirse como una pirámide compuesta por cuatro capas. La capa fundamental se centra en el diseño de subsistemas, que implementan funciones principales de sistema. La capa de clases especifica la arquitectura de objetos global, y la jerarquía de clases requerida para implementar un sistema. La capa de mensajes indica cómo debe ser realizada la colaboración entre objetos, y la capa de responsabilidades identifica las operaciones y atributos que caracterizan cada clase.

Al igual que el AOO, existen diferentes métodos de DOO. UML es un intento de proporcionar una aproximación simple al DOO, que se aplica en los dominios de aplicaciones. UML y otros métodos, aproximan el proceso de diseño mediante dos niveles de abstracción —diseño de subsistemas (arquitectura) y diseño de objetos individuales—.

Durante el diseño del sistema, la arquitectura del sistema orientado a objetos se desarrolla. Además del desarrollo de sistemas, de **sus** interacciones y de **su** colocación dentro de las capas arquitectónicas, el diseño de sistemas considera la componente de interacción con el usuario, una componente de administración de tareas y una componente de manejo de datos. Estas componentes de subsistemas proporcionan la infraestructura de diseño, que permite a la aplicación operar efectivamente. El proceso de diseño de objetos se centra en la descripción de estructuras de datos, que usan los atributos de clase, los algoritmos que usan las operaciones y los mensajes que permiten colaboraciones entre objetos relacionados.

Los patrones de diseño permiten al diseñador crear la arquitectura de diseño integrando componentes reusables. La programación OO extiende el modelo de diseño a un dominio de ejecución. Un lenguaje de programación OO se usa para traducir las clases, atributos, operaciones y mensajes, de manera que puedan ejecutarse por la máquina.

REFERENCIAS

[BEN99] Bennett, S., S. McRobb y R. Farmer, *Object Oriented System Analysis and Design Using UML*, McGraw Hill, 1999.

[BIH92] Bihari, T., y P. Gopinath, «Object-Oriented Real-Time Systems: Concepts and Examples», *Computer*, vol. 25, n.º 12, Diciembre 1992, pp. 25-32.

[BOO94] Booch, G., *Object-Oriented Analysis and Design*, 2.ª ed., Benjamin Cummings, 1994.

[BOO99] Booch, G., I. Jacobson y J. Rumbaugh, *The Unified Modelling Language User Guide*, Addison-Wesley, 1999.

[BRO91] Brown, A.W., *Object-Oriented Databases*, McGraw-Hill, 1991.

[BUS96] Buschmann, F., et al., *A System of Patterns: Pattern Oriented System Architecture*, Wiley, 1996.

[COA91] Coad, P., y E. Yourdon, *Object-Oriented Design*, Prentice-Hall, 1991.

[COX85] Cox, B., «Software Ics and Objective-C», *UnixWorld*, primavera de 1985.

[DAV95] Davis, A., «Object-Oriented Requirements to Object-Oriented Design: An Easy Transition?», *J. Systems Software*, vol. 30, 1995, pp. 151-159.

[DOU99] Douglass, B., *Real-Time UML: Developing Efficient Objects for Embedded Systems*, Addison-Wesley, 1999.

[GAM95] Gamma, E., et al., *Design Patterns*, Addison-Wesley, 1995.

[GOL83] Goldberg, A., y D. Robson, *Smalltalk-80: The Languuge and Its Implementation*, Addison-Wesley, 1983.

[JAC92] Jacobson, I., *Object-Oriented Software Engineering*, Addison-Wesley, 1992.

[JAC99] Jacobson, I., G. Booch y J. Rumbaugh, *Unified Software Development Process*, Addison-Wesley, 1999.

[MEY90] Meyer, B., *Object-Oriented Software Construction*, 2.ª ed., Prentice-Hall, 1988.

[PRE95] Pree, W., *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

[RUM91] Rumbaugh, J., et al., *Object-Oriented Modelling and Design*, Prentice-Hall, 1991.

[RUM99] Rumbaugh, J., I. Jacobson y G. Booch, *The Unified Modelling Language Reference Manual*, Addison-Wesley, 1999.

[RAO94] Rao. B.A., *Object-Oriented Databases: Technology, Applications and Products*, McGraw-Hill, 1994.

[TAY92] Taylor, D.A., *Object-Oriented Information System*, Wiley, 1992.

[WIR90] Wirfs-Brock, R., B. Wilkerson y L. Weiner, *Designing Object-Oriented Software*, Prentice-Hall, 1990.

PROBLEMAS Y PUNTOS A CONSIDERAR

21.1. La pirámide de diseño para el DOO difiere, de alguna manera, de la descrita para el diseño del software convencional (Capítulo 13). Vea las diferencias y semejanzas de ellas dos.

21.2. ¿Cómo difieren el DOO y el estructurado? ¿Qué aspectos de estos dos métodos de diseño son los mismos?

21.3. Revise los cinco criterios para la modularidad eficaz examinados en la Sección 22.1.2. Usando el enfoque de diseño descrito posteriormente en el capítulo, demuestre cómo se logran estos cinco criterios.

21.4. Seleccione uno de los métodos de DOO presentados en la Sección 22.1.3, y prepare un tutorial de una hora para su clase. Asegúrese de mostrar todas las convenciones gráficas de modelado que sugiere el autor.

21.5. Seleccione un método de DOO no presentado en la Sección 22.1.3, (por ejemplo, HOOD), y prepare un tutorial de una hora para **su** clase. Asegúrese de mostrar todas las convenciones gráficas de modelado que sugiere el autor.

21.6. Analice cómo los casos de uso pueden servir como una fuente importante de información para el diseño.

21.7. Investigue un entorno de desarrollo IGU, y muestre cómo el componente de interacción hombre-máquina se implementa en el mundo real. ¿Qué patrones de diseño se ofrecen y cómo se usan?

21.8. La gestión de tareas en sistemas OO puede ser muy compleja. Realice alguna investigación sobre métodos de DOO, para sistemas en tiempo real (por ejemplo, [BIH92] o [DOU99]) y determine cómo la gestión de tareas se realiza dentro de este contexto.

21.9. Examine cómo el componente de manejo de datos se implementa en un entorno de desarrollo OO típico.

21.10. Escriba un artículo de dos o tres página sobre bases de datos orientadas a objetos, y analice cómo pueden usarse, para desarrollar el componente de gestión de datos.

21.11. ¿Cómo reconoce un diseñador las tareas que deben ser recurrentes?

21.12. Aplique el enfoque del DOO examinado anteriormente en este capítulo, para diseccionar el diseño del sistema *HogarSeguro*. Defina todos los subsistemas relevantes, y desarrolle el diseño de objetos para las clases importantes.

21.13. Aplique el enfoque del DOO examinado en este capítulo al sistema SSRB descrito en el problema 12.13.

21.14. Describa un juego de vídeo **y** aplique el enfoque del DOO examinado en este capítulo, para representar su diseño.

21.15. Usted es responsable del desarrollo de un sistema de correo electrónico a implementarse sobre una red de PC. **El** sistema de e-mail permitirá a los usuarios escribir cartas dirigidas a otro usuario o a una lista de direcciones específica. Las cartas pueden ser enviadas, copiadas, almacenadas, etc. **El** sistema de correo electrónico hará uso de las capacidades de procesamiento de texto existentes para escribir las cartas. Usando esta descripción como punto de partida, derive un conjunto de requisitos, y aplique las técnicas de DOO, para crear un diseño de alto nivel, para el sistema de correo electrónico.

21.16. Una nación ubicada en una pequeña isla ha decidido construir un sistema de control de tráfico aéreo (CTA) para **su** único aeropuerto. El sistema se especifica como sigue:

> Todos los aviones que aterrizan en el aeropuerto deben tener un transmisor que transmita el tipo de avión **y los** datos del vuelo en un paquete, con formato de alta densidad, a la estación de **CTA** de tierra. La estación de **CTA** de tierra puede interrogar a un avión, para solicitar información específica y almacenarla en una base de datos de aviones. Se crea un monitor de gráficos por ordenador, a partir de la información almacenada, y se la muestra a **un** controlador de tráfico. El monitor se actualiza cada 10 segundos. Toda la información es analizada, para determinar si existen «situaciones peligrosas». **El** controlador de tráfico aéreo puede interrogar la base de datos, para buscar información específica acerca de cualquier avión que se muestre en pantalla.

Usando el DOO, cree un diseño para el sistema de CTA. No intente implementarlo.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Además de las muchas referencias de este capítulo, los libros de Gossain y Graham *(Object modelling and Design Strategies,* SIGS Books, 1998), Meyer (Object-Oriented Design Through Heuristics, Addison-Wesley, 1996), y Walden, Jean-Marc Nerson *(Seamless Object-Oriented Software Architecture: Analisis and Design of Reliable Systems*, Prentice-Hall, 1995) cubren con bastante detalle el DOO. Fowler (*Refactoring: improving the Design of Existing Code,* Addison-Wesley, 1999) dirige el uso de técnicas orientadas a objetos para rediseñar y reconstruir programas antiguos con el fin de mejorar su calidad de diseño.

Muchos libros de diseño orientado a objetos publicados recientemente enfatizan UML. El lector seriamente interesado en aplicar UML a **su** trabajo debe adquirir [BOO99], [RUM99] y [JAC99]. Además, muchos de los libros referidos en la sección *otras lecturas* y *fuentes de información* del Capítulo 21 también se centran en el diseño con un nivel de detalle considerable.

El uso de patrones de diseño para el desarrollo de software orientado a objetos tiene importantes implicaciones para la ingeniería del software basada en componentes, la reutilización en general y la calidad global de los sistemas resultantes. Además de [BUS96] y [GAM95], muchos libros recientes dedican **sus** páginas al mismo tema, como los siguientes:

Ambler, S.W., *Process Patterns: Building Large-Scale Systems Using Object Technology,* Cambridge University Press, 1999.

Coplien, J.O., D.C. Schmidt, *Pattern Languages of Program Design,* Addison-Wesley, 1995.

Fowler, M., *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1996.

Grand, M., *Patterns in Java*, vol. 1, John Wiley, 1998.

Langr, J., *Essential Java Style: Patterns for Implementation*, Prentice-Hall, 1999.

Larman, C., *Applying Unl and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice-Hall, 1997.

Martin, R.C., et al., *Pattern Languages of Program Design 3*, Addison-Wesley, 1997.

Rising, L., y J. Coplien (eds.), *The Patterns Handbook: Techniques, Strategies, and Applications*, SIGS Books, 1998.

Pree, W., *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

Preiss, B., *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*, John Wiley, 1999.

Vlissides, J., Pattem Hatching: *Design Patterns Applied*, Addison-Wesley, 1998.

Vlissides, J.M., J.O. Coplien y N. Kerth, *Pattern Languages of Program Design 2*, Addison-Wesley, 1996.

Cientos de libros de programación orientada a objetos (POO) han sido publicados. Una muestra de libros específicos en lenguajes de POO:

C++: Cohoon, J.P., *C++ Program Design: An Introduction to Programming and Object-Oriented Design*, McGraw-Hill, 1998.

Barclay, K., y J. Savage, *Object-Oriented Design With C++*, Prentice-Hall, 1997.

Eiffel: Thomas, P., y R. Weedon, *Object-Oriented Programming in Eiffel*, Addison-Wesley, 1997.

Jezequel, J.M., *Ohject-Oriented Software Engineering With Eiffel*, Addison-Wesley, 1996.

Java: Coad, P., M. Mayfield y J. Kem, *Java Design: Building Better Apps and Applets*, 2.ª ed., Prentice-Hall, 1998.

Lewis, J., y W. Loftus, *Java Software Solutions: Foundations of Program*, Addison-Wesley, 1997.

Smalltalk: Sharp, A., *Smalltalk by Example: The Developer's Guide*, McGraw-Hill, 1997.

LaLonde, W.R., y J.R. Pugh, *Programming in Smalltalk*, Prentice-Hall, 1995.

Los libros que cubren temas de DOO, mediante el uso de dos o más lenguajes de programación, proporcionan una idea y comparación de las capacidades de los lenguajes. Algunos títulos son:

Drake, C., *Ohject-Oriented Programming With C++ and Smalltalk*, Prentice Hall, 1998.

Joyner, I., *Objects Unencapsulated: Java, Eiffel and C++*, Prentice Hall, 1999.

Zeigler, B.P., *Objects and Systems: Principled Design With Implementations in C++ and Java*, Springer Verlag, 1997.

Una amplia variedad de fuentes de información sobre diseño orientado a objetos, así como temas relacionados, están disponibles en internet. Una lista actualizada de referencias a sitios (páginas) web, que pueden ser relevantes al DOO, pueden encontrarse en **http://www.pressman5.com**

CAPÍTULO 23 PRUEBAS ORIENTADAS A OBJETOS

EL objetivo de las pruebas, expresado de forma sencilla, es encontrar el mayor número posible de errores con una cantidad razonable de esfuerzo, aplicado sobre un lapso de tiempo realista. A pesar de que este objetivo fundamental permanece inalterado para el software orientado a objetos, la naturaleza de los programas OO cambian las estrategias y las tácticas de prueba.

Puede argumentarse que, conforme el AOO y el DOO maduran, una mayor reutilización de patrones de diseño mitigarán la necesidad de pruebas intensivas en los sistemas OO. La verdad es justo lo contrario. Binder [BIN94b] lo analiza, cuando afirma que:

> Cada reutilización es un nuevo contexto de uso y es prudente repetir las pruebas. Parece probable que se necesitarán menos pruebas para obtener una alta fiabilidad en sistemas orientados a objetos.

La prueba de los sistemas OO presentan un nuevo conjunto de retos al ingeniero del software. La definición de pruebas debe ser ampliada para incluir técnicas que descubran errores (revisiones técnicas formales), aplicadas para los modelos de AOO y DOO. La integridad, compleción y consistencia de las representaciones OO deben ser evaluadas conforme se construyen. Las pruebas de unidad pierden mucho de su significado, y las estrategias de integración cambian de modo significativo. En resumen, las estrategias y tácticas de prueba deben tomarse en cuenta para las características propias del software OO.

VISTAZO RÁPIDO

¿Qué es? La arquitectura de software orientado a objetos se manifiesta en una serie de subsistemas organizados por capas, que encapsulan clases que colaboran entre sí. Cada uno de estos elementos del sistema (subsistemas y clases), realizan funciones que ayudan a alcanzar requerimientos del sistema. Es necesario probar un sistema OO, en una variedad de niveles diferentes, en un esfuerzo para descubrir errores, que deben ocurrir cuando las clases colaboran con otras entre sí, y los subsistemas se comunican por medio de las capas arquitectónicas.

¿Quién lo hace? Las pruebas orientadas a objetos se realizan por ingenieros de software y especialistas en pruebas.

¿Por qué es importante? Se debe ejecutar el programa antes de que llegue al cliente, con la intención específica de descubrir todos los errores, de manera que el cliente no experimente la frustración asociada con un producto de baja calidad. Con el propósito de encontrar el mayor número posible de errores, las pruebas deben conducirse sistemáticamente. y los casos de prueba deben ser designados mediante técnicas disciplinadas.

¿Cuáles son los pasos? Las pruebas OO son estratégicamente similares a las pruebas de sistemas convencionales. pero tácticamente diferentes. Ya que los modelos de análisis y diseño OO son similares en estructura y contenido al programa OO, las «pruebas» comienzan con la revisión de estos modelos. Una vez que se ha generado el código, las pruebas OO comienzan «en lo pequeño. con las pruebas de clases. Existen pruebas diseñadas para probar las operaciones de las clases y examinar si los errores existen cuando una clase colabora con otras. Así como las clases se integran para formar un subsistema basado en hilos, basado en usos y pruebas de agrupamiento, junto con los enfoques basados en fallos, se aplican para probar exhaustivamente las clases colaboradoras. Por último, los casos de uso (desarrollados como parte del modelo de análisis OO) se usan para descubrir errores en el software a nivel de validación.

¿Cuál es el producto obtenido? Se diseñan y documentan un conjunto de casos de prueba, diseñados para probar clases, sus colaboraciones y comportamientos; se definen los resultados esperados y se registran los resultados reales

¿Cómo puedo estar seguro de que lo he hecho correctamente? Cuando comienzan las pruebas, se cambia su punto de vista. ¡Intenta «romper» el software! Diseñar los casos de prueba de una forma disciplinada y revisar los casos de prueba que se crean con meticulosidad.

23.1 AMPLIANDO LA VISIÓN DE LAS PRUEBAS

La construcción del software orientado a objetos comienza con la creación de los modelos de análisis y diseño (Capítulos 21 y 22). Debido a la naturaleza evolutiva del paradigma de ingeniería de software OO, estos modelos comienzan como representaciones, relativamente informales, de los requisitos del sistema, y evolucionan en modelos detallados de clases, conexiones y relaciones de clases, el diseño del sistema y el diseño de objetos (incorporando un modelo de conectividad de objetos por medio de mensajes). En cada etapa, los modelos pueden probarse, en un intento de descubrir errores, antes de su propagación a la siguiente iteración.



Debido a su capacidad para detectar y corregir defectos en productos de trabajo anteriores, las revisiones técnicas son por lo menos tan importantes, en el control de los costes y la planificación, como las pruebas.
Steve McConnell

Puede argumentarse que la revisión de los modelos de análisis y diseño OO son especialmente útiles, ya que las mismas estructuras semánticas (por ejemplo, clases, atributos, operaciones, mensajes), aparecen en los niveles de análisis, diseño y codificación. Por consiguiente, un problema en la definición de los atributos de las clases que se descubren durante el análisis evitará efectos laterales que pueden ocurrir si el problema no se descubriera hasta el diseño o implementación (o incluso la siguiente iteración del análisis).

Por ejemplo, considérese una clase en la que el número de atributos se definen durante la primera iteración del AOO. Un atributo externo (extraño), se agrega a la clase (debido al malentendido del dominio de problema). Se especifican dos operaciones para manipular el atributo. Se realiza una revisión y un experto en el dominio señala el error. Eliminando el atributo irrelevante en esta etapa, los problemas y esfuerzos innecesarios se evitan durante el análisis:

1. Pueden haberse generado subclases especiales para adoptar (alojar) el atributo innecesario o excepciones a él. El trabajo involucrado en la creación de subclases no necesarias se ha evitado.
2. Una mala interpretación de la definición de la clase puede conducir a relaciones de clases incorrectas o irrelevantes.
3. El comportamiento del sistema o sus clases pueden caracterizarse inadecuadamente, para alojar el atributo irrelevante.

Si el error no se descubre durante el análisis y se propaga más allá, los siguientes problemas pueden ocurrir (y tendrán que evitarse con una nueva revisión) durante el diseño:

1. La localización impropia de la clase a un subsistema y/o tareas puede ocurrir durante el diseño del sistema.
2. El trabajo del diseño innecesario tendrá que ser recuperado, para crear el diseño procedimental, para las operaciones que afecten al atributo innecesario.
3. El modelo de mensajes (mensajería) será incorrecto (debido a que deben diseñarse mensajes para las operaciones innecesarias).

Si el error permanece sin detectarse durante el diseño y pasa a la actividad de codificación, se gastará un esfuerzo considerable para generar el código que implementa un atributo innecesario, dos operaciones innecesarias, mensajes que controlan comunicaciones entre objetos, y muchos otros aspectos relacionados. Además, la prueba de la clase absorberá más tiempo del necesario. Una vez que se encuentra el problema en su totalidad, debe llevarse a cabo la modificación del sistema, teniendo siempre presentes los posibles efectos colaterales producidos por el cambio.



Existe un viejo dicho que dice «cortar por lo sano». Si pierde tiempo revisando los modelos de AOO y DOO, después lo ganará.

Durante las etapas finales de su desarrollo, los modelos de AOO y de DOO proporcionan información substancial acerca de la estructura y comportamiento del sistema. Por esta razón, estos modelos deben estar sometidos a una revisión rigurosa, antes de la generación de código.

Todos los modelos orientados a objetos deben ser probados (en este contexto, el término «prueba» se utiliza para incorporar revisiones técnicas formales), para asegurar la exactitud, compleción y consistencia [MGR94], dentro del contexto de la sintaxis, semántica y pragmática del modelo [LIN94].

23.2 PRUEBAS DE LOS MODELOS DE AOO Y DOO

Los modelos de análisis y diseño no pueden probarse en el sentido convencional, ya que no pueden ejecutarse. **Sin** embargo, se pueden utilizar las revisiones técnicas formales (Capítulo 8) para examinar la exactitud y consistencia de ambos modelos de análisis y diseño.

23.2.1. Exactitud de los modelos de AOO y DOO

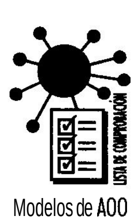
La notación y sintaxis que se utiliza para representar los modelos de análisis y diseño se corresponderá con el método específico de análisis y diseño, elegido para el proyecto. Por consiguiente, la exactitud sintáctica se juzga en el uso apropiado de la simbología; cada modelo se revisa para asegurarse de que se han mantenido las convenciones propias del modelado.

Durante el análisis y diseño, la exactitud semántica debe juzgarse basada en la conformidad del modelo con el dominio del problema en el mundo real. Si el modelo refleja con exactitud el mundo real (al nivel de detalle apropiado a la etapa de desarrollo en la que se revisa el modelo), entonces es semánticamente correcto. Para determinar si el modelo en realidad refleja el mundo real, debe presentarse a expertos en el dominio del problema, quienes examinarán las definiciones de las clases y sus jerarquías, para detectar omisiones o ambigüedades. Las relaciones entre clases (conexiones de instancia) se evalúan para determinar si reflejan con exactitud conexiones del mundo real[1].

23.2.2. Consistencia de los modelos de AOO y DOO

La consistencia de los modelos de AOO y DOO debe juzgarse «considerando las relaciones entre entidades dentro del modelo. Un modelo inconsistente tiene representaciones en una parte, que no se reflejan correctamente en otras partes del modelo» [MGR94].

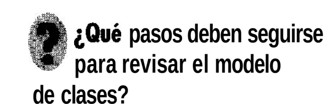
Para evaluar la consistencia, se debe examinar cada clase y sus conexiones a otras clases. Un modelo clase-responsabilidad-colaboración (CRC), y un diagrama objeto-relación pueden utilizarse para facilitar esta actividad. Como se comentó en el Capítulo 21, el modelo CRC se compone de una tarjeta índice CRC. Cada tarjeta CRC muestra el nombre de la clase, sus responsabilidades (operaciones) y sus colaboradores (otras clases a las que se envían mensajes y de las cuales depende para el cumplimiento de sus responsabilidades). Las colaboraciones implican una serie de relaciones (por ejemplo, conexiones), entre clases del sistema OO. El modelo objeto-relación proporciona una representación gráfica de las conexiones entre clases. Toda esta información se puede obtener del modelo de AOO (Capítulo 21).



Modelos de AOO

Para evaluar el modelo de clases, se recomiendan los siguientes pasos [MGR94]:

1. *Revisar el modelo CRC y el modelo objeto-relación.* Realizar un control cruzado, para asegurarse de que todas las colaboraciones implicadas en el modelo de AOO hayan sido representadas adecuadamente.


¿Qué pasos deben seguirse para revisar el modelo de clases?

2. *Inspeccionar la descripción de cada tarjeta CRC, para determinar si alguna responsabilidad delegada es parte de la definición del colaborador.* Por ejemplo, considérese una clase definida para un sistema de control de punto de venta, llamada venta a crédito. Esta clase tiene una tarjeta CRC, que se ilustra en la Figura 23.1.
   Para esta colección de clases y colaboraciones, se pregunta si alguna responsabilidad (por ejemplo, *leer la tarjeta de crédito)* se cumple si se delega al colaborador nombrado *(Tarjeta de crédito).* Esto significa que la clase *Tarjeta de crédito* posee una operación para ser leída. En este caso, la respuesta es «Sí». El objeto-relación se recorre,. para asegurarse de que todas las conexiones son válidas.

Referencia cruzada
Sugerencias adicionales para conducir una revisión del modelo CRC se presentan en el **Capítulo 21.**

3. Invertir la conexión para asegurarse de que cada colaborador que solicita un servicio recibe las peticiones de una fuente razonable. Por ejemplo, si la clase *Tarjeta de crédito* recibe una petición de *cantidad de compra* de la clase *Venta a crédito,* existirá un problema. *Tarjeta de crédito* no reconoce la *cantidad de compra.*

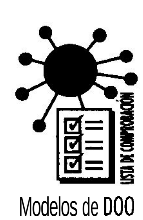
[1] Los casos de uso poseen un valor incalculable, en el seguimiento de los modelos de análisis y diseño, frente a los escenarios del mundo real para el sistema OO.

Nombre de la clase: venta a crédito	
Tipo de la clase: evento de transacción	
Características de la clase: intangible, atómica, secuencial, permanente, protegida	
Responsabilidades:	**Colaboradores:**
Leer tarjeta de crédito	Tarjeta de crédito
Obtener autorización	Autorización de crédito
Enviar por correo cantidad de compra	Etiqueta de producto
	Vendedor
	Archivo auditor
Generar factura	Factura

FIGURA 23.1. Un ejemplo de tarjeta índice CRC usada para revisión.

4. *Utilizando las conexiones invertidas, ya examinadas en el paso 3, determinar si otras clases se requieren y si las responsabilidades se han repartido adecuadamente entre las clases.*
5. *Determine si las responsabilidades muy solicitadas, deben combinarse en una sola responsabilidad.* Por ejemplo, *leer tarjeta de crédito* **y** *obtener autorización* ocurren en cada situación. Por consiguiente, se pueden combinar en la responsabilidad *validar petición de crédito,* que incorpora obtener el número de tarjeta de crédito, **y** conseguir la autorización.
6. *Se aplican iterativamente los pasos 1 a 5 para cada clase, y durante cada evolución del modelo de AOO.*

Una vez que se crea el modelo de DOO (Capítulo 22), deben llevarse a cabo también las revisiones del diseño del sistema **y** del diseño de objetos. El diseño del sistema describe el producto arquitectónico global, los subsistemas que componen el producto, la manera en que los subsistemas se asignan a los procesadores, la asignación de clases a subsistemas **y** el diseño de la interfaz de usuario. El diseño de objetos presenta los detalles de cada clase, **y** las actividades de mensajería necesarias para implementar las colaboraciones entre clases.



El diseño de sistema se revisa examinando el modelo objeto-comportamiento desarrollado durante el AOO, y la correspondencia necesaria del comportamiento del sistema, frente a los subsistemas diseñados para lograr este comportamiento.

La concurrencia **y** asignación de tareas también se revisan dentro del contexto del comportamiento del sistema. Los estados de comportamiento del sistema se evalúan para determinar cuáles existen concurrentemente. Los escenarios o situaciones de los casos de uso se utilizan para validar el diseño de la interfaz de usuario.

El modelado de objetos debe probarse frente a la red objeto-relación, para asegurarse de que todos los objetos de diseño contienen los atributos **y** operaciones necesarias **y** para implementar las colaboraciones que se definieron para cada tarjeta CRC. Además, la especificación detallada de las operaciones (por ejemplo, los algoritmos que implementan a las operaciones), se revisan usando técnicas de inspección convencionales.

23.3 ESTRATEGIAS DE PRUEBAS ORIENTADAS A OBJETOS

La estrategia clásica para la prueba de software de ordenador, comienza con «probar lo pequeño» **y** funciona hacia fuera haciendo «probar lo grande». Siguiendo la jerga de la prueba de software (Capítulo 18), se comienza con las pruebas de unidad, después se progresa hacia las pruebas de integración **y** se culmina con las pruebas de validación del sistema. En aplicaciones convencionales, las pruebas de unidad se centran en las unidades de programa compilables más pequeñas —el subprograma (por ejemplo, módulo, subrutina, procedimiento, componente)—. Una vez que cada una de estas unidades han sido probadas individualmente, se integran en una estructura de programa, mientras que se ejecutan una serie de pruebas de regresión para descubrir errores, debidos a las interfaces entre los módulos **y** los efectos colaterales producidos por añadir nuevas unidades. Por último, el sistema se comprueba como un todo para asegurarse de que se descubren los errores en requisitos.

Cita:
El mejor probador no es el que encuentra la mayoría de los errores...; el mejor es el que repara la mayoría de ellos.
Cem Kaner et al.

23.3.1. Las pruebas de unidad en el contexto de la OO

Cuando se considera el software orientado a objetos, el concepto de unidad cambia. La encapsulación conduce a la definición de clases **y** objetos. Esto significa que cada clase **y** cada instancia de una clase (objeto), envuel-

ven atributos (datos) y operaciones (también conocidos como métodos o servicios), que manipulan estos datos. En vez de probar un módulo individual, la unidad más pequeña comprobable es la clase u objeto encapsulado. Ya que una clase puede contener un número de operaciones diferentes, y una operación particular debe existir como parte de un número de clases diferentes, el significado de la unidad de prueba cambia drásticamente.

No se puede probar más de una operación a la vez (la visión convencional de la unidad de prueba), pero sí como parte de una clase. Para ilustrar esto, considérese una jerarquía.de clases, en la cual la operación X se define para la superclase y se hereda por algunas subclases. Cada subclase utiliza la operación X, pero se aplica en el contexto de los atributos y operaciones privadas que han sido definidas para la subclase. Ya que el contexto en el que la operación X se utiliza varía de manera sutil, es necesario para probar la operación X en el contexto de estas subclases. Esto significa que probar la operación X en vacío (la aproximación de la prueba de unidades tradicionales) es inefectiva en el contexto orientado a objetos.



CLAVE

La prueba de software OO es equivalente al módulo de pruebas unitarias para el software convencional. No es recomendable comprobar operaciones por separado.

La prueba de clases para el software OO es el equivalente de las pruebas de unidad para el software convencional*. A diferencia de las pruebas de unidad del software convencional que tienden a centrarse en el detalle algontmico de un módulo y de los datos que fluyen a través de la interfaz del módulo, la prueba de clases para el software OO se conduce mediante las operaciones encapsuladas por la clase y el comportamiento de la clase.

23.3.2. Las pruebas de integración en el contexto OO

Ya que el software orientado a objetos no tiene una estructura de control jerárquico, las estrategias convencionales de integración descendente (top-down) y ascendente (bottom-up) tienen muy poco significado. En suma, la integración de operaciones una por una en una clase (la aproximación de la integración incremental convencional), a menudo es imposible por la «interacción directa e indirecta de los componentes que conforman la clase» [BER93].

Existen dos estrategias diferentes para las pruebas de integración de los sistemas OO [BIN94a]. El primero, las *pruebas basadas en hilos*, integran el conjunto de clases requeridas, para responder una entrada o suceso al sistema. Cada *hilo* se integra y prueba individualmente. Las pruebas de regresión se aplican para asegurar que no ocurran efectos laterales. La segunda aproximación de integración, la *prueba basada en el uso*, comienza la construcción del sistema probando aquellas clases (llamadas clases independientes), que utilizan muy pocas (o ninguna) clases servidoras. Después de que las clases independientes se prueban, esta secuencia de pruebas por capas de *clases dependientes* continúa hasta que se construye el sistema completo. A diferencia de la integración convencional, el uso de drivers y stubs como operaciones de reemplazo, debe evitarse siempre que sea posible.

C VE

La estrategia de integración de pruebas OO se centra en grupos de clases que colaboran o se comunican de la misma manera.

La prueba de agrupamiento [MGR94] es una fase en las pruebas de integración de software OO. Aquí, un agrupamiento de clases colaboradoras (determinadas por la revisión de los modelos CRC y objeto-relación), se prueba diseñando los casos de prueba, que intentan revelar errores en las colaboraciones.

23.3.3. Pruebas de validación en un contexto OO

Al nivel de sistema o de validación, los detalles de conexiones de clases desaparecen. Así como la validación convencional, la validación del software OO se centra en las acciones visibles al usuario y salidas reconocibles desde el sistema. Para ayudar en la construcción de las pruebas de validación, el probador debe utilizar los casos de uso (Capítulo 20), que son parte del modelo de análisis. Los casos de uso proporcionan un escenario, que tiene una gran similitud de errores con los revelados en los requisitos de interacción del usuario.

Referencia cruzada

Aparentemente, todos los métodos de pruebas de caja negra discutidos en el Capítulo 17 son aplicables a la OO.

Los métodos de prueba convencionales de caja negra pueden usarse para realizar pruebas de validación. Además, los casos de prueba deben derivarse del modelo de comportamiento del objeto y del diagrama de flujo de sucesos, creado como parte del AOO.

[2] Los métodos de diseño para pruebas de caso, para las clases **OO**, se discuten de las Secciones 23.4 a ***23.6***.

23.4 DISEÑO DE CASOS DE PRUEBA PARA SOFTWARE OO

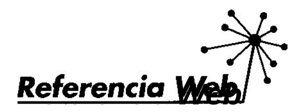
Los métodos de diseño de casos de prueba para software orientado a objetos continúan evolucionando. Sin embargo, una aproximación global al diseño de casos de prueba OO ha sido definida por Bernard [BER93]:

1. Cada caso de prueba debe ser identificado separadamente, y explícitamente asociado con la clase a probar.
2. Debe declararse el propósito de la prueba.
3. Debe desarrollarse una lista de pasos a seguir, como consecuencia de la prueba, pero además debe contener [BER93]:
   a. definición de una lista de estados, específicos para el objeto a probar.
   b. una lista de mensajes y operaciones, que se ejercitarán como consecuencia de las pruebas.
   c. una lista de excepciones, que pueden ocurrir conforme el objeto se comprueba.
   d. una lista de condiciones externas (por ejemplo, los cambios en el ambiente externo al software, que debe existir para conducir apropiadamente las pruebas).
   e. información adicional, que ayudará a la comprensión e implementación de la prueba.

A diferencia del diseño de pruebas convencional, que se conduce mediante una visión entrada-proceso-salida de software, o el detalle algorítmico de los módulos individuales, la prueba orientada a objetos se enfoca en las secuencias de operaciones de diseño apropiadas para probar los estados de una clase.

23.4.1. Implicaciones de los conceptos de OO al diseño de casos de prueba

Como ya se ha visto, la clase es el objetivo del diseño de casos prueba. Debido a que los atributos y operaciones se encapsulan, las operaciones de prueba fuera de la clase son generalmente improductivas. A pesar de que la encapsulación es un concepto de diseño esencial para la OO, puede crear un obstáculo cuando se hacen las pruebas. Como menciona Binder [BIN94a], «la prueba requiere informes del estado abstracto y concreto del objeto». La encapsulación puede dificultar un poco la obtención de esta información. A menos que se proporcionen operaciones incorporadas para conocer los valores para los atributos de la clase, una imagen instantánea del estado del objeto puede ser difícil de adquirir.



Una colección excelente de publicaciones, fuentes y bibliografías de pruebas OO puede encontrarse en **www.rbsc.com**

La herencia también conduce a retos adicionales para el diseñador de casos de prueba. Ya se ha dicho que cada nuevo contexto de uso requiere repetir la prueba, aún y cuando se haya logrado la reutilización. Además, la herencia múltiple[3] complica mucho más las pruebas, incrementando el número de contextos para los que se requiere la prueba [BIN94a]. Si las subclases instanciadas de una superclase se utilizan dentro del mismo dominio de problema, es probable que el conjunto de casos de prueba derivados de la superclase puedan usarse para la prueba de las subclases. De cualquier manera, si la subclase se utiliza en un contexto enteramente diferente, los casos prueba de la superclase serán escasamente aplicables, y tendrá que diseñar un nuevo conjunto de pruebas.

23.4.2. Aplicabilidad de los métodos convencionales de diseño de casos de prueba

Los métodos de «caja blanca» descritos en el Capítulo 17 pueden aplicarse a las operaciones definidas para una clase. Técnicas como el camino básico, pruebas de bucle o técnicas de flujo de datos pueden ayudar a asegurar que se ha probado cada sentencia de la operación. De cualquier modo, la estructura concisa de muchas operaciones de clase provoca que algunos defiendan que el esfuerzo aplicado a la prueba de «caja blanca» debe ser redirigido adecuadamente a las pruebas, a un nivel de clase.

Los métodos de prueba de «caja negra» son tan apropiados para los sistemas OO, como lo son para los sistemas desarrollados utilizando los métodos convencionales de ingeniería de software. Como se dijo al principio del capítulo, los casos de uso pueden proporcionar datos Útiles en el diseño de pruebas de «caja negra», y pruebas basadas en estados [AMB95].

23.4.3. Pruebas basadas en errores[4]

El objetivo de las pruebas basadas en errores dentro de un sistema OO, es diseñar pruebas que tengan una alta probabilidad de revelar fallos. Ya que el producto o sistema debe adaptarse a los requerimientos del cliente, la

[3] Concepto de DOO que debe usarse con extrema precaución.

[4] Las Secciones 23.4.3 a *23.4.6* han sido adaptadas de un artículo' de Brian Marick, divulgado en el grupo de noticias de internet **comp.testing.** Esta adaptación se ha incluido con el permiso del autor. Para adentrarse más en la discusión de este tema, véase [MAR94].

planificación preliminar requerida para llevar a cabo la prueba basada en fallos comienza con el modelo de análisis. El probador busca fallos posibles (por ejemplo, los aspectos de implementación del sistema que pueden manifestarse en defectos). Para determinar si existen estos fallos, los casos de prueba se diseñan para probar el diseño o código.

CLAVE

La estrategia consiste en hacer hipótesis de una serie de posibles fallos, y luego conducir las pruebas para comprobar las hipótesis.

Considérese un ejemplo simple[5]. Los ingenieros de software generalmente cometen errores en los límites del problema. Por ejemplo, cuando se prueba una operación SQRT que genera errores para números negativos, se sabe probar los límites: un número negativo cercano al cero y el cero mismo. El «cero mismo» comprueba si el programador ha cometido un error como:

```
If( X > 0 ) calcular_la_raíz_cuadrada( );
```

En lugar de lo correcto

```
If( X >= 0 ) calcular-la-raíz-cuadrada( );
```

Como otro ejemplo, considérese la expresión booleana siguiente:

```
If (a && !b || c )
```



Ya que la prueba basada en fallos se da en un nivel detallada, se reservo mejor para las operaciones y clases que son críticas a sospechosas.

Las pruebas de multicondición y técnicas relacionadas examinan las faltas posibles con toda seguridad en esta expresión, como, por ejemplo,

&& debería de ser ||

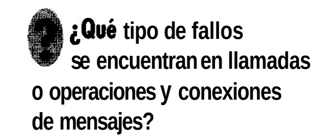
! se ignoró donde se necesitaba

Y debería haber un paréntesis alrededor de !b ||c

Para cada error probable, se diseñan casos de prueba, que forzarán a la condición incorrecta a fallar. En la expresión anterior, (*a=0, b=0, c=0*) forzarán a que la expresión se evalúe *falsa*. Si el && hubiera sido ||, el código hubiera dado el resultado incorrecto y el control habría seguido la rama errónea.

Desde luego que la efectividad de estas técnicas depende de cómo los probadores perciben el «fallo probable». Si los fallos verdaderos en un sistema OO se perciben como «improbables», entonces esta aproximación no es en realidad mejor que la técnica de pruebas aleatorias. De cualquier manera, si los modelos de análisis y diseño pueden proporcionar la visión de lo que parece andar mal, entonces las pruebas basadas en errores pueden encontrar un número significativo de errores con esfuerzos relativamente pequeños.

Las pruebas de integración buscan fallos probables en operaciones o mensajes de conexión. Tres tipos de fallos se encuentran en este contexto: resultados inesperados, uso incorrecto de operaciones/mensajes, invocaciones incorrectas. Para determinar fallos probables cuando las funciones (operaciones) se invocan, se debe examinar el comportamiento de la operación.



¿Qué tipo de fallos se encuentran en llamadas o operaciones y conexiones de mensajes?

Las pruebas de integración se aplican tanto a atributos como a operaciones. Los «comportamientos» de un objeto se definen por los valores que se asignan a sus atributos. Las pruebas deben verificar los atributos para determinar si se obtienen valores apropiados para los distintos tipos de comportamientos de los objetos.

Es importante hacer notar que las pruebas de integración intentan encontrar los errores en el objeto cliente, no en el servidor. Dicho en términos comunes, el enfoque de las pruebas de integración es determinar si el error existe en el código de invocación, no en el código invocado. La invocación a operaciones se utiliza como una pista, una forma de encontrar los requerimientos de la prueba que validen el código de invocación.

23.4.4. El impacto de la programación OO en las pruebas

Hay distintas maneras en que la programación orientada a objetos puede tener un impacto en las pruebas. Dependiendo del enfoque de la POO.

- Algunos tipos de fallos se vuelven menos probables (no vale la pena probar).
- Algunos tipos de fallos se vuelven más probables (vale la pena probar).
- Aparecen algunos tipos nuevos de fallos.



Si usted desea y espera que un programa funcione, será más probable que vea un programa funcionando —extrañará fallos—.
Cem Kaner et al.

[5] El código presentado en ésta y en las siguientes secciones utiliza la sintaxis de C++. Para mayor información, véase cualquier buen libro de C++.

Cuando se invoca una operación es difícil decir exactamente qué código se ejecuta. Esto es, la operación debe pertenecer a una de muchas clases. Incluso, puede ser difícil determinar el tipo exacto o clase de un parámetro. Cuando el código lo acceda puede obtenerse un valor inesperado. La diferencia puede entenderse considerando la llamada a una función convencional:

x = func(y);

Para el software convencional, el probador debe considerar todos los comportamientos atribuidos a *func* y nada más. En un contexto **OO,** el probador debe considerar los comportamientos de ***base::func( )***, de ***derivada::func( )***, y así sucesivamente. Cada vez que *func* se invoca, el probador debe considerar la unión de todos los comportamientos distintos. Esto es más fácil si se siguen prácticas de diseño OO adecuadas, y se limitan las diferencias entre superclases y subclases (en el lenguaje de C++, estas se llaman ***clases base*** y ***clases derivadas).***

La aproximación a las pruebas para clases derivadas y base es esencialmente la misma. La diferencia es de contabilidad.

Probar las operaciones de clases es análogo a probar código que toma un parámetro de la función y luego la invoca. La herencia es una manera conveniente de producir operaciones polimórficas. En la llamada, lo que importa no es la herencia, sino el polimorfismo. La herencia hace que la búsqueda de los requisitos de prueba sea más directa.

En virtud de la construcción y arquitectura de software, ¿existen algunos tipos de fallos más probables para un sistema **OO,** y otros menos probables? la respuesta es «**sí**». Por ejemplo, a causa de que las operaciones OO son generalmente más pequeñas, se tiende a gastar más tiempo en la integración ya que existen más oportunidades de errores de integración. **Así** que los errores de integración se vuelven más probables.

23.4.5. Casos de prueba y jerarquía de clases

Como se explicó previamente en este capítulo, la herencia no obvia la necesidad de probar todas las clases derivadas. De hecho, puede complicar el proceso de prueba.



Aunque se haya comprobado bastante una clase base, tendrá que comprobar las clases derivadas de ella.

Considérese la situación siguiente. Una clase base contiene operaciones heredadas **y** redefinidas. Una clase derivada redefine ***operaciones redefinidas*** para funcionar en un contexto local. Existe la pequeña duda de que la *derivada::redefinida( )* debe ser comprobada, porque representa un nuevo diseño y nuevo código. Pero, ¿la *derivada::heredada( )* debe ser recomprobada?

Si la *derivada:heredada( )* invoca a ***redefinida*** y el comportamiento de ***redefinida*** cambia, *derivada::heredada( )* podría manejar erróneamente el nuevo comportamiento. De este modo, se necesitan nuevas pruebas aunque el diseño y el código no hayan cambiado. Es importante hacer notar, sin embargo, que solo un subconjunto de todas las pruebas para *derivada::heredada( )* deben rehacerse. Si parte del diseño y codificación para ***heredada*** no depende de ***redefinida*** (por ejemplo, no la llama ni llama ningún código que indirectamente la llame), ese código no necesita ser recomprobado en la clase derivada.

Base:redefinida( ) y derivada::redefinida( ) son dos operaciones diferentes con especificaciones e implementaciones diferentes. Cada una debería tener un conjunto de requisitos de prueba, derivadas de la especificación e implementación. Aquellos requisitos prueban errores probables: fallos de integración, fallos de condición, fallos de límites y **así** sucesivamente. Pero las operaciones es probable que sean similares por lo que el conjunto de requisitos de pruebas se solapan. Cuanto mejor sea el diseño **OO,** el solapamiento será mayor. Es necesario desarrollar las nuevas pruebas, solo para aquellos requisitos de *derivada::redefinida( )* que no fueron satisfechas por pruebas de *base::redefinida( )*.

Para concluir, las pruebas de *base::redefinida( )* se aplican a los objetos de la clase derivada. Las entradas de las pruebas deben ser apropiadas para las clases base y derivada, pero los resultados esperados pueden diferir en la clase derivada.

23.4.6. Diseño de pruebas basadas en el escenario

Las pruebas basadas en los errores no localizan dos tipos de errores: (1) especificaciones incorrectas y *(2)* interacción entre subsistemas. Cuando ocurren errores asociados con especificaciones erróneas, el producto no realiza lo que el cliente desea. Puede que haga cosas incorrectas, o puede omitir funcionalidad importante. Pero en cualquier circunstancia, la calidad (cumplimiento de requisitos) se sacrifica. Los errores asociados con la interacción de subsistemas ocurren cuando el comportamiento de un subsistema crea circunstancias, (por ejemplo, sucesos, flujo de datos) que causan que el otro subsistema falle.

Las pruebas basadas en el escenario se concentran en lo que el usuario hace, no en lo que el producto hace. Esto significa capturar las tareas (por medio de los casos de uso) que el usuario tiene que hacer, después aplicarlas a ellas y a sus variantes como pruebas.



La prueba basada en el escenario descubrirá errores que ocurren cuando cualquier actor interoctúa con el software OO.

Los escenarios revelan errores de interacción. Pero para llevar a cabo esto, los casos de prueba deben ser más complejos y realistas que las pruebas basadas en los errores. Las pruebas basadas en el escenario tienden a validar subsistemas en una prueba sencilla (los usuarios no se limitan al uso de un subsistema a la vez).

A manera de ejemplo, considérese el diseño de pruebas basadas en el escenario, para un editor de texto. Utilícense los siguientes casos:

Caso de uso: Prepara la versión final.

Aspectos de fondo: No es inusual imprimir el borrador «final», leerlo y descubrir algunos errores incómodos que no eran tan obvios en la imagen de la pantalla. Este caso de uso describe la secuencia de pasos que ocurren cuando esto pasa.

1. **Imprimir el documento entero.**
2. **Rondar dentro del documento, cambiar algunas páginas.**
3. **Al momento en que cada página se cambia, se imprime.**
4. **Algunas veces se imprime una serie de páginas.**

Este escenario describe dos elementos: una prueba y unas necesidades específicas del usuario. Las necesidades del usuario son obvias: (1) un método para imprimir una sola página y (2) un método para imprimir un rango de páginas. Hasta donde va la prueba, existe la necesidad de editar después de imprimir (así como lo contrario). El probador espera descubrir que la función de impresión causa errores en la función de edición; esto es, que las dos funciones de software sean totalmente independientes.

Caso de uso: Impresión de una nueva copia.

Aspectos de fondo: Se pide una copia reciente. Debe ser impresa:

1. **Abrir el documento**
2. **Imprimirlo.**
3. **Cerrar el documento.**



Aunque lo prueba basada en el escenario tiene su mérito, encontrará uno recompenso mayor revisondo los casos de uso cuando se desarrollan durante el AOO.

Una vez más, la aproximación de las pruebas es relativamente obvia. Excepto que el documento no apareció fuera de ningún lugar. Fue creado en una tarea anterior. ¿La tarea anterior afecta a esta?

En los editores modernos, los documentos registran como fueron impresos por última vez. Por omisión, se imprimen de la misma forma la siguiente vez. Después del escenario *preparar la versión final*, con solo seleccionar «imprimir» en el menú y presionando el botón «imprimir» en la caja de diálogo, se conseguirá que la última página corregida se imprima de nuevo. **Así** que, de acuerdo con el editor, el escenario correcto debería ser como el siguiente:

Caso de uso: Imprimir una nueva copia.

1. **Abrir el documento.**
2. **Seleccionar «imprimir» en el menú.**
3. **Comprobar si se imprime un rango de páginas; si es así, presionar para «imprimir» el documento entero.**
4. **Presionar en el botón de impresión.**
5. **Cerrar el documento.**

Pero este escenario indica una especificación potencial de error. El editor no hace lo que el usuario razonablemente espera. Los clientes generalmente pasarán por alto la opción de rango de páginas del paso 3 en el ejemplo anterior. Se incomodarán cuando enciendan la impresora y encuentren una página cuando ellos querían 100. Los clientes fastidiados significan errores de especificación.

Un diseñador de casos de prueba debe olvidar la dependencia en un diseño de pruebas, pero es probable que el problema aparezca durante las pruebas. El probador tendría entonces que lidiar con la respuesta probable, «¿Esa es la forma como se supone que debe funcionar?».

23.4.7. Las estructuras de pruebas superficiales y profundas

La estructura superficial se refiere a la estructura visible al exterior de un programa OO. Esto ***es,*** la estructura que es inmediatamente obvia al usuario final. En vez de llevar a cabo funciones, los usuarios de muchos sistemas OO deben de proveerse de objetos para manipular de alguna forma. Pero sin importar la interfaz, las pruebas aún se basan en las tareas de los usuarios. Capturar estas tareas involucra comprensión, observación, y conversar con usuarios representativos **(y** tantos usuarios no representativos como valga la pena considerar).

Debe haber alguna diferencia en detalle con seguridad. Por ejemplo, en un sistema convencional con una interfaz orientada a comandos, el usuario debe usar la lista de comandos como una lista de control de pruebas. Si no existen escenarios de prueba para ejercitar un comando, las pruebas probablemente pasaron por alto algunas tareas del usuario (o la interfaz contiene comandos inútiles). En una interfaz basada en objetos, el verificador debe usar la lista de todos los objetos, como una lista de control de pruebas.



lo estructura de pruebas se da en dos niveles: (1) pruebas que validan la estructura visible por el usuario final, (2) pruebas diseñadas para validar la estructura interna del programa.

Las mejores pruebas se dan cuando el diseñador observa al sistema de una manera nueva o poco convencional. Por ejemplo, si el sistema o producto tiene una interfaz basada en comandos, pruebas más completas se darán si el diseñador de casos supone que las operaciones son independientes de los objetos. Hacer preguntas como «¿Querrá el usuario usar esta operación —que se aplica sólo al objeto **scanner**— mientras trabaja con la impresora?». Cualquiera que sea el estilo de la interfaz, el diseño de casos de prueba que ejercita la estructura superficial debe usar ambos objetos y operaciones, como pistas que conduzcan a las tareas desapercibidas.

La estructura profunda se refiere a los detalles técnicos de un programa OO. Esto es, la estructura que se comprende examinando el diseño y/o el código. La verificación de la estructura profunda está diseñada para ejercitar dependencias, comportamientos y mecanismos de comunicación, que han sido establecidos como parte del diseño del objeto y del sistema (Capítulo 22) de software OO.

Los modelos de análisis y diseño se utilizan como una base para la verificación de la estructura profunda. Por ejemplo, el diagrama objeto-relación o el diagrama de colaboración de subsistemas describe colaboraciones entre objetos y subsistemas, que no pueden ser visibles externamente.

El diseño de casos de prueba entonces se pregunta: «¿Se ha capturado (como una prueba) alguna tarea que pruebe la colaboración anotada en el diagrama objeto relación, o en el diagrama de colaboración de subsistemas? Si no es así, ¿por qué no?»

El diseño de representaciones de jerarquías de clases proporciona una visión de la estructura de herencia. La estructura jerárquica se utiliza en la verificación basada en errores. Considérese la situación en la cual una operación llamada ***llamar_a( )*** tiene un solo argumento, y ese argumento es una referencia a la clase base. ¿Qué ocurrirá cuando ***llamar_a( )*** pase a la clase derivada? ¿Cuáles serán las diferencias en comportamiento que puedan afectar a tal función? Las respuestas a estas preguntas pueden conducir al diseño de pruebas especializadas.

23.5 MÉTODOS DE PRUEBA APLICABLES AL NIVEL DE CLASES

En el Capítulo 17 se mencionó que la prueba de software comienza «en lo pequeño» y lentamente progresa hacia la prueba «a grande». La prueba «en pequeño», se enfoca en una sola clase y los métodos encapsulados por ella. La verificación y partición al azar son métodos que pueden usarse para ejercitar a una clase durante la prueba OO [KIR94].

23.5.1. La verificación al azar para clases OO

A manera de ilustraciones sencillas de estos métodos, considérese una aplicación bancaria en la cual una clase **cuenta** contiene las siguientes operaciones: ***abrir, configurar, depositar, retirar, consultar saldo, resumen, Limitecrédito y cerrar*** [KIR94]. Cada una de estas operaciones debe aplicarse a **cuenta,** pero algunas limitaciones (por ejemplo, la cuenta debe ser abierta antes de que otras operaciones puedan aplicársele, y cerrada después de que todas las operaciones hayan sido completadas) están implícitas en la naturaleza del probletna. Aún con estas limitaciones, existen muchas combinaciones de operaciones. El registro de operaciones mínima de una instancia de **cuenta** incluye las siguientes operaciones:

Abrir – coniigurar – depositar – retirar – cerrar.



El número de combinaciones posibles para uno prueba aleatoria puede crecer mucha. Una estrategia similar poro lo pruebas de arrays ortogonales (Capítulo 17), puede usarse poro mejorar lo eficiencia de los pruebas.

Esto representa la secuencia de verificación mínima para una cuenta. De cualquier modo, pueden existir una amplia variedad de combinaciones de operaciones posibles, dentro de esta secuencia:

Abrir – configurar – depositar – [depositar | retirar | consultar saldo | resumen | LimiteCrédito]^n – retirar – cerrar

Pueden generarse una variedad de secuencias de operaciones diferentes al azar. Por ejemplo,

Prueba r_1: abrir – configurar – depositar – consultar saldo – resumen – retirar – cerrar.
Prueba r_2: abrir – coniigurar – depositar – retirar – depositar – consuitar saldo – LímiteCrédito – retirar – cerrar.

Estas y otras pruebas de orden aleatorio se realizan para probar diferentes registros de operaciones de instancias de clases.

23.5.2. Prueba de partición al nivel de clases

La prueba de partición reduce el número de casos de prueba requeridos para validar la clase, de la misma forma que la partición equivalente (Capítulo 17) para software convencional. Las entradas y salidas se clasifican, y los casos de prueba se diseñan, para validar cada categoría. Pero ¿cómo se derivan las categorías de partición?

¿Qué opciones de pruebas están disponibles a nivel de clases?

La partición basada en estados clasifica las operaciones de clase basada en su habilidad de cambiar el

estado de la clase. Una vez más, considerando la clase **cuenta,** operaciones de estado incluyen a *depositar* y *retirar,* y considerando que las operaciones de no-estado incluyen a *consultar saldo, resumen* y *LímiteCrédito,* las pruebas se diseñan de manera que las operaciones que cambian el estado, y aquellas que no lo cambian, se ejerciten separadamente. Entonces:

Prueba p_1: *abrir – configurar – depositar – depositar – retirar – retirar – cerrar.*
Prueba p_2: *abrir – configurar – depositar – resumen – Límitecrédito – retirar – cerrar.*

La prueba p_1 cambia el estado, mientras que la prueba p_2 ejercita las operaciones que no cambian el estado (excepto por las necesarias de la secuencia mínima de prueba).

La partición basada en atributos clasifica las operaciones de clase basada en los atributos que ellas usan. Para la clase **cuenta,** los atributos **saldo** y **LímiteCrédito** pueden usarse para definir particiones. Las operaciones se dividen en tres particiones: (1) Operaciones que utilizan **LimiteCrédito**, (2) operaciones que modifican **Límitecrédito,** y (3) operaciones que no utilizan o modifican **Límitecrédito.** Las secuencias de prueba se diseñan por cada partición.

La partición basada en categorías clasifica las operaciones de la clase basadas en la función genérica que cada una lleva a cabo. Por ejemplo, las operaciones en la clase **cuenta** pueden clasificarse en operaciones de **inicialización (abrir, configurar),** operaciones computacionales **(depositar, retirar),** consultas **(saldo, resumen, Límite-Crédito)** y operaciones de terminación **(cerrar).**

23.6 DISEÑO DE CASOS DE PRUEBA INTERCLASES

El diseño de casos de prueba se vuelve más complicado cuando la integración del sistema OO comienza. Es en esta etapa en que la verificación de colaboraciones entre clases comienza. Para ilustrar «la generación de casos de prueba interclases» [KIR94], se expande el ejemplo de la aplicación bancaria introducida en la Sección 23.5, para incluir las clases y colaboraciones de la Figura 23.2. La dirección de las flechas en la Figura indica que se invocan como consecuencia de colaboraciones implícitas a los mensajes.

Así como la verificación de clases individuales, la verificación de colaboraciones de clases puede completarse aplicando métodos de partición y al azar, así como pruebas basadas en el escenario y pruebas de comportamiento.

23.6.1. Prueba de múltiples clases

Kirani y Tsai [KIR94] sugieren la secuencia siguiente de pasos, para generar casos de prueba aleatorios para múltiples clases:

1. Para cada clase cliente, utilice la lista de operaciones de clase, para generar una serie de secuencias de pruebas al azar. Las operaciones enviarán mensajes a las otras clases servidoras.
2. Para cada mensaje que se genere, determine la clase colaboradora y la operación correspondiente en el objeto servidor.
3. Para cada operación en el objeto servidor (invocada por mensajes enviados por el objeto cliente), determine los mensajes que transmite.
4. Para cada uno de los mensajes, determine el siguiente nivel de operaciones que son invocadas, e incorpore éstas a la secuencia de pruebas.

Para ilustrar [KIR94], considérese una secuencia de operaciones para la clase **banco** relativa a una clase **ATM** (Figura 23.2):

verifCuenta – verifNIP – [[verifPoliza – reqRetirar] | reqDepositar | reqInfoCuenta]^n

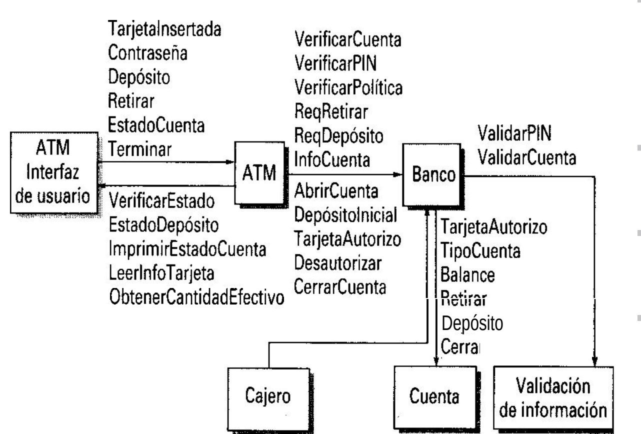


FIGURA 23.2. Grafo de colaboraciones de clase para una aplicación bancaria [KIR94].

Un caso de prueba aleatorio para la clase **banco** podría ser:

Prueba r_3 = *verifCuenta – verifNIP – reqDepositar*

Con la finalidad de considerar a los colaboradores involucrados en esta prueba, los mensajes asociados con cada una de las operaciones mencionadas en el caso de prueba r_3 deben ser tomados en cuenta. **Banco** debe colaborar con **infoValidación** para ejecutar *verifCuenta* y *verifNIP.* **Banco** debe colaborar con **cuenta** para ejecutar *reqDepositar.* De aquí que un nuevo caso de prueba, que ejercite estas colaboraciones, es:

Prueba r_4 = $verifCuenta_{Banco}$ [$validCuenta_{infoValidación}$]
– $VerifNIP_{Banco}$ – [$ValidNIP_{infoValidación}$]
– reqDepositar – [$Depositar_{Cuenta}$]

La aproximación para la prueba de partición de múltiples clases es similar a la aproximación usada para la

prueba de partición de clases individuales. La manera como una sola clase se particiona se discutió en la Sección 23.4.5. De cualquier modo, la secuencia de pruebas se extiende para incluir aquellas operaciones que se invocan por los mensajes a clases colaboradoras. Una aproximación alternativa basa las pruebas por partición en las interfaces de una clase en particular. Haciendo referencia a la Figura 23.2, la clase **banco** recibe mensajes de las clases **ATM y Cajero.** Los métodos incluidos en **banco** pueden probarse particionándolos en aquellos que sirven a **ATM y** aquellos que sirven a **Cajero.** La partición basada en estados (Sección 23.4.9), puede usarse para refinar aún más las particiones.

23.6.2. Prueba derivada de los modelos de comportamiento

En el Capítulo 21 se discutió el uso del diagrama de transición de estados, como el modelo que representa el comportamiento dinámico de una clase. El DTE (Diagrama de transición de estados) para una clase puede usarse para ayudar a derivar una secuencia de pruebas, que ejercitarán el comportamiento dinámico de la clase (y aquellas clases que colaboran con ella). La Figura 23.3 [KIR94] ilustra una DTE para la clase **cuenta** explicada con anterioridad[6]. Con referencia a la Figura, las transiciones iniciales se mueven por los estados ***cuenta*** vacía *y* configura ***cuenta.*** Un retiro final y cierre causan que la clase cuenta haga transiciones a los estados ***nohacetrabajo*** de cuenta *y* cuenta ***muerta,*** respectivamente.

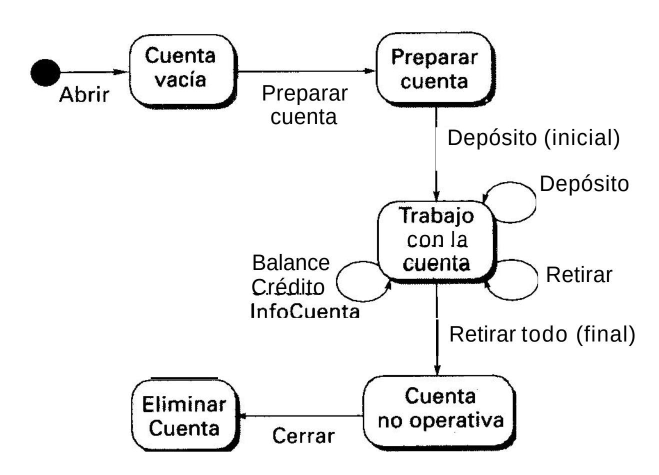


FIGURA 23.3. Diagrama de transición de estados para la clase cuenta [KIR94].

Las pruebas a diseñarse deben alcanzar una cobertura de todos los estados [KIR94]. Eso significa que las secuencias de operaciones deben causar que la clase **cuenta** haga transiciones por todos los estados:

Prueba s_1: abrir –preparar cuenta – depositar (inicial) – retiro (final) – cerrar

Nótese que esta secuencia es idéntica a la secuencia mínima de pruebas, discutida en la Sección 23.5.1. Si se agregan secuencias de prueba adicionales a la secuencia mínima:

Prueba s_2: abrir – preparar cuenta – depositar(inicial)– saldo – crédito – retiro (final) – cerrar.

Prueba s_3: preparar cuenta – depositar (inicial)– retiro – infoCuenta – retiro (final) cerrar.

Se pueden derivar aún más casos de prueba, para asegurarse de que todos los comportamientos para la clase han sido adecuadamente ejercitados. En situaciones en las que el comportamiento de la clases, resulte en una colaboración con una o más clase se utilizan múltiples DTEs para registrar el flujo de comportamientos del sistema.

El modelo de estado puede ser recorrido «primero a lo ancho» [MGR94]. En este contexto, ***primero a lo ancho*** implica que un caso de prueba valida una sola transición **y** después, cuando se va a verificar una nueva transición, se utilizan sólo las transiciones previamente verificadas.

Considérese el objeto **tarjeta–de–crédito** de la Sección 23.2.2. El estado inicial de **tarjeta–de–crédito** es indefinido (es decir, no se ha proporcionado un número de tarjeta de crédito). Una vez leída la tarjeta de crédito durante una venta, el objeto asume un estado ***definido;*** esto significa que los atributos **número tarjeta y fecha expiración,** se definen junto con identificadores específicos del banco. La tarjeta de crédito ***se envía,*** cuando se envía la autorización, *y* es ***aprobada*** cuando la autorización se recibe. La transición de **tarjeta–de–crédito** de un estado a otro puede probarse generando casos de prueba, que hagan que la transición ocurra. Un enfoque «primero a lo ancho» en este tipo de pruebas, puede no validar ***el envío*** antes de que se ejercite ***indefinida*** y ***definida.*** Si lo hiciera, haría uso de transiciones que no han sido verificadas con anterioridad, y violaría el criterio «primero a lo ancho».



Uno extensa colección de ((consejos sobre pruebas OO» (incluyendo muchos referencias útiles) puede encontrarse en:
www.kinetica.com/ootips/

[6] La simbologia UML se utiliza para el DTE que se ilustra en la Figura 23.3. Difiere ligeramente de la simbología usada para los DTEs en la parte tres de este libro.

RESUMEN

El objetivo global de la verificación orientada a objetos -encontrar el máximo número de errores con un mínimo de esfuerzo—, es idéntico al objetivo de prueba del software convencional. Pero la estrategia y tácticas para la prueba OO difieren de modo significativo. La visión de verificación se amplía, para incluir la revisión de ambos modelos de diseño y de análisis.

En resumen, el enfoque de prueba se aleja del componente procedimental (el módulo) hacia la clase. Ya que los modelos de análisis y diseño OO y el código fuente resultante se acoplan semánticamente, la prueba (a manera de revisiones técnicas formales) comienza en estas actividades de ingeniería. Por esta razón, la revisión de los métodos CRC, objeto-relación y objeto-comportamiento, pueden verse como una primera etapa de prueba.

Una vez que la POO ha sido concluida, las pruebas de unidad se aplican a cada clase. El diseño de pruebas para una clase utiliza una variedad de métodos: pruebas basadas en errores, las pruebas al azar y las pruebas por partición. Cada uno de estos métodos ejercita las operaciones encapsuladas por la clase. Las secuencias de pruebas se diseñan para asegurarse de que las operaciones relevantes se ejerciten. El estado de la clase representada por los valores de sus atributos se examina, para determinar si persisten errores.

La prueba de integración puede llevarse a cabo utilizando una estrategia basada en hilos o basada en el uso. La estrategia basada en hilos integra el conjunto de clases, que colaboran para responder a una entrada o suceso. Las pruebas basadas en el uso construyen el sistema en capas, comenzando con aquellas clases que no usan clases servidoras. Los métodos de diseño de integración de casos de prueba pueden usar pruebas al azar y por partición. En suma, las pruebas basadas en el escenario y las pruebas derivadas de los modelos de comportamiento pueden usarse para verificar una clase y sus colaboraciones. Una secuencia de pruebas registra el flujo de operaciones, a través de las colaboraciones de clases.

La prueba de validación de sistemas OO está orientada a caja negra y puede completarse aplicando los mismos métodos de prueba de caja de negra discutidos para el software convencional. Sin embargo, las pruebas basadas en el escenario dominan la validación de sistemas **OO,** haciendo que el caso de uso sea el conductor primario para las pruebas de validación.

REFERENCIAS

[AMB95] Ambler, S., «Using Use Cases», ***Software Development***, Julio de 1995, pp. 53-61.

[BER93] Berard, E.V., ***Essays on Ohject-Oriented Software Engineering,*** vol. 1, Addison-Wesley, 1993.

[BIN94a] Binder, R.V., «Testing Object-Oriented Systems: A Status Report», ***American Programmer,*** vol. 7, n.° 4, Abril de 1994, pp. 23-28.

[BIN94b] Binder, R.V., «Object-Oriented Software Testing», ***CACM,*** vol. 37, n.° 9, Septiembre de 1994, p. 29.

[CHA93] DeChampeaux, D., D. Lea y P. Faure, ***Object-Oriented System Development,*** Addison-Wesley, 1993.

[FIC92] Fichman, R., y C. Kemerer, «Object-Oriented and conceptual Design Methodologies», ***Computer,*** vol. 25, n.° 10, Octubre de 1992, pp. 22-39.

[KIR94] Kirani, S., y W.T. Tsai, «Specification and Verification of Object-Oriented Programs», ***Technical Report TR 94-96,*** Computer Science Department, University of Minnesota. Diciembre de 1994.

[LIN94] Lindland, O.I., et al., «Understanding Quality in Conceptual Modeling», ***IEEE Software,*** vol. 11, n.° 4, Julio de 1994, pp. 42-49.

[MAR94] Marick, B., ***The Craft of Software Testing,*** Prentice Hall. 1994.

[MGR94] McGregor, J.D., y T.D. Korson, «Integrated Object-Oriented Testing and Development Processes», ***CACM,*** vol. 37, n." 9, Septiembre de 1994, pp. 59-77.

PROBLEMAS Y PUNTOS A CONSIDERAR

23.1. Describa con sus propias palabras por qué la clase es la más pequeña unidad razonable para las pruebas dentro de un sistema OO.

23.2. ¿Por qué debemos probar de nuevo las subclases instanciadas de una clase existente si ésta ya ha sido probada por completo? ¿Podemos usar los casos de prueba diseñados para dicha clase?

23.3. ¿Por qué debe comenzar la «prueba» con las actividades de AOO y DOO?

23.4. Derive un conjunto de tarjetas índice CRC para ***HogarSeguro*** y ejecute los pasos señalados en la Sección 23.2.2 para determinar si existen inconsistencias.

23.5. ¿Cuál es la diferencia entre las estrategias basadas en hilos y aquellas estrategias basadas en uso para las pruebas de integración? ¿Cómo cabe la prueba de agrupación en ellas?

23.6. Aplique la prueba aleatoria y la de partición a tres clases definidas en el diseño del sistema ***HogarSeguro*** producido por usted en el Problema 22.12.

Produzca casos de prueba que indiquen las secuencias de operaciones que se invocarán.

23.7. Aplique la prueba de múltiples clases y las pruebas derivadas del modelo de comportamiento al diseño de *HogarSeguro.*

23.8. Obtenga pruebas usando los métodos señalados en los Problemas 23.6 y 23.7 para el sistema **SSRB** descrito en el Problema 22.13.

23.9. Obtenga pruebas usando los métodos señalados en los Problemas 23.6 y 23.7 para el juego de vídeo considerado en el Problema 22.14.

23.10. Obtenga pruebas usando los métodos señalados en los Problemas 23.6 y 23.7 para el sistema de e-mail considerado en el Problema 22.15.

23.11. Obtenga pruebas usando los métodos señalados en los Problemas 23.6 y 23.7 para el sistema ATC considerado en el Problema 22.16.

23.12. Obtenga cuatro pruebas adicionales usando cada uno de los métodos señalados en los Problemas 23.6 y 23.7 para la aplicación bancaria presentada en las Secciones 23.5 y 23.6.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

La literatura para la prueba OO es relativamente escasa, aunque se ha expandido algo en años recientes. Binder *(Testing Ohject-Oriented Systems: Models, Patterns, and Tools,* Addison-Wesley, 2000) ha escrito el tratamiento más extenso del tema publicado hasta la fecha. Siegel y Muller *(Object Oriented Software Testing: A Hierarchical Approach,* Wiley, 1996) propusieron una estrategia práctica de prueba para sistemas OO. Marick *(The Craft of Software Testing: Subsystem Testing Including Ohject-Based and Ohject-Oriented Testing,* Prentice Hall, 1995) cubre la prueba tanto para software convencional como para OO.

Antologías de publicaciones sobre prueba OO han sido editadas por Kung et al. *(Testing Object-Oriented Software,* IEEE Computer Society, 1998) y Braude *(Ohject Oriented Analysis, Design and Testing: Selected Readings,* IEEE Computer Society, 1998). Estos tutoriales de IEEE proporcionan una interesante perspectiva histórica en el desarrollo de prueba OO.

Jorgenesen *(Software Testing: A Cruftsman 's Approuch,* CRC Press, 1995) y McGregor y Sykes *(Ohject-Oriented Software Development,* Van Nostrand Reinhold, 1992) presenta capítulos dedicados al tema. Beizer *(Black-Box Testing,* Wiley, 1995) analiza una variedad de métodos de diseño de casos prueba los cuales son apropiados en un contexto OO.

Binder *(Testing Object-Oriented Systems,* Addison-Wesley, 1996) y Marick [MAR94] presenta tratamientos detallados de prueba OO. En resumen, muchas de las fuentes anotadas para el Capítulo 17 son en general aplicables a la prueba OO.

Una amplia variedad de fuentes de información de prueba orientada a objetos y temas relacionados se encuentran disponibles en intemet. Una lista reciente a sitios (páginas) web que son relevantes a la prueba OO puede encontrarse en **http://www.pressman5.com**

CAPÍTULO

24 MÉTRICAS TÉCNICAS PARA SISTEMAS ORIENTADOS A OBJETOS

EN secciones anteriores de este libro, se mencionó que las métricas y mediciones son componentes clave para cualquier disciplina de ingeniería —y la ingeniería de software orientada a objetos no es una excepción—. Desgraciadamente, el uso de métricas para sistemas OO ha progresado mucho más despacio que'el uso de otros métodos OO. Ed Berard [BER95] mencionó la ironía de las mediciones, cuando declaraba que:

> La gente involucrada en el software parece tener una relación amor-odio con las métricas. Por una parte, menosprecian y desconfían de cualquier cosa que parezca o suene a medición. Son rápidos, bastante rápidos para señalar las «imperfecciones», en los argumentos de cualquiera que hable acerca de las mediciones a los productos de software, procesos de software, y (especialmente) personas involucradas en software. Por otra parte, esta misma gente parece no tener problemas al identificar qué lenguaje de programación es el mejor, las estupideces que los administradores hacen para «arruinar» proyectos, y la metodología de trabajo en qué situaciones.

La «relación amor-odio» que Berard declara es real. **Más** aun, como los sistemas OO se encuentran cada vez más implantados, es esencial que los ingenieros en software posean mediciones cuantitativas, para la evaluación de calidad de diseños, a niveles arquitectónicos **y** de componentes.

Estas mediciones permiten al ingeniero evaluar el software al inicio del proceso, haciendo cambios que reducirán la complejidad y mejorarán la viabilidad, a largo plazo, del producto final.

VISTAZO RÁPIDO

¿Qué es? Construir software OO ha sido una actividad de ingeniería, que confía más en el juicio, folklore y referencias cualitativas, que en la evaluación cuantitativa. Las métricas OO se han introducido para ayudar a un ingeniero del software a usar el análisis cuantitativo, para evaluar la calidad en el diseño antes de que un sistema se construya. El enfoque de métricas OO está en la clase —la piedra fundamental de la arquitectura OO—, la arquitectura **OO,** las clases y subsistemas que conforman la arquitectura, las operaciones y atributos que constituyen una clase. El comprobador necesita las referencias cuantitativas, que le ayudarán en la selección de casos de prueba y sus objetivos. Las métricas técnicas proporcionan una base desde la cual el análisis, el diseño y la verificación pueden conducirse de manera **más** objetiva, y ser evaluados más cuantitativamente.

¿Quién lo hace? Los ingenieros del software se ayudan de las métricas para construir software de mayor calidad.

¿Por qué es importante? Como se expuso en el vistazo rápido del Capítulo 19, la evaluación cualitativa de software debe complementarse con el análisis cuantitativo. Un ingeniero del software necesita un criterio objetivo para ayudarse a conducir el diseño de

¿Cuáles son los pasos? El primer paso en el proceso de medición es deducir las mediciones de software y métricas que pudieran ser apropiadas para la representación del software en consideración. Después, se recolectan los datos requeridos para aplicar las métricas formuladas. Una vez computados, se analizan basándose en orientaciones preestablecidas y en datos anteriores. Los resultados del análisis son interpretados para obtener una visión inherente a la calidad del software, y los resultados de la interpretación conducen a la modificación de resultados de trabajo deducidos del análisis, diseño, codificación o prueba.

¿Cuál es el producto obtenido? Las métricas de software que se calculan mediante los datos recolectados de los modelos de análisis y diseño, código fuente y casos de prueba.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Debe establecer los objetivos de las mediciones antes de que la recolección de datos comience, definiendo cada métrica OO de manera concreta. Definir solamente algunas métricas y después utilizarlas, para obtener una vista inherente a la calidad de un producto de ingeniería de software.

24.1 EL PROPÓSITO DE LAS MÉTRICAS ORIENTADAS A OBJETOS

Los objetivos primarios para las métricas orientadas a objetos no son diferentes de aquellos de las métricas desarrolladas para el software convencional:

- para entender mejor la calidad del producto.
- para evaluar la efectividad del proceso.
- para mejorar la calidad del trabajo llevado a cabo al nivel del proyecto.

Cada uno de estos objetivos es importante, pero para el ingeniero de software la calidad del producto debe ser lo más importante. Pero ¿cómo medir la calidad de un sistema OO? ¿Qué características del modelo de diseño pueden y deben evaluarse para determinar si el sistema será fácil de implementar, fácil de verificar, simple de modificar y, lo más importante, aceptable para los usuarios finales? Estas preguntas serán contestadas en la parte restante del capítulo.

24.2 CARACTERÍSTICAS DISTINTIVAS DE LAS MÉTRICAS ORIENTADAS A OBJETOS

Las métricas para cualquier producto de ingeniería son reguladas por las características únicas del producto. Por ejemplo, sería inútil o sin sentido contar las millas por galón de consumo para un automóvil eléctrico. La métrica es firme para los automóviles convencionales (es decir, impulsados por sistemas de combustión interna a gasolina) pero no se aplica cuando el modo de propulsión cambia radicalmente. El software orientado a objetos es fundamentalmente diferente del software desarrollado con el uso de métodos convencionales. Por esta razón, las métricas para sistemas OO deben ser afinadas a las características que distinguen al software OO del software convencional.

Berard [BER95] define cinco características que regulan las métricas especializadas: Localización, encapsulación, ocultamiento de información, herencia y técnicas de abstracción de objetos. Cada una de estas características se discute brevemente en las secciones siguientes[1].



Referencia Web

Una extensión de búsqueda para métricas OO la puedes obtener en: **www.objenv.com /cetus / oo_metrics.html**

24.2.1. Localización

La localización es una característica del software que indica la manera en que la información se concentra en un programa. Por ejemplo, los métodos convencionales para la descomposición funcional organizan la información en torno a las funciones que son típicamente implementadas como módulos procedimentales. Los métodos manejados por datos localizan la información en tomo a estructuras de datos específicas. En el contexto **OO,** la información se concentra por la encapsulación de datos y procesos dentro de los límites de una clase u objeto.

Ya que el software convencional enfatiza la función como un mecanismo de localización, las métricas de software se han enfocado a la estructura interna o funciones de complejidad (por ejemplo, longitud del módulo, cohesión o complejidad ciclomática), o a la manera como las funciones se conectan entre sí (acoplamiento de módulos).

Referencia cruzada

Métricos técnicas pam software convencional se describen en el Capítulo 19.

Puesto que la clase es la unidad básica de un sistema **OO,** la localización se basa en los objetos. Por esta razón, las métricas deben aplicarse a la clase (objeto), como a una entidad completa. En suma, la relación entre operaciones (funciones) y clases no es necesariamente uno a uno. Por lo tanto, las métricas que reflejan la manera en la que las clases colaboran deben ser capaces de acomodarse a relaciones uno a muchos y muchos a uno.

24.2.2. Encapsulación

Berard [BER95] define la encapsulación como el «empaquetamiento (o ligamento) de una colección de elementos. Ejemplos de bajo nivel de encapsulación [para software convencional] incluyen registros y matrices, [y] subprogramas (por ejemplo, procedimientos, funciones, subrutinas y párrafos), son mecanismos de nivel medio para la encapsulación.»

Referencia cruzada

Los conceptos básicos de diseño se describen en el Capítulo 13. Su aplicación al software OO se discute en el Capítulo 20.

Para los sistemas **OO,** la encapsulación engloba las responsabilidades de una clase, incluyendo sus atributos (y otras clases para objetos agregados) y operacio-

[1] Este estudio ha sido adaptado de [BER95]

nes, y los estados de la clase, definidos por valores de atributos específicos.

La encapsulación influye en las métricas, cambiando el enfoque de las mediciones de un módulo simple, a un paquete de datos (atributos) y módulos de proceso (operaciones).

En suma, la encapsulación eleva la medición a un nivel de abstracción más alto.

Por ejemplo, más adelante en este capítulo se introducirán las métricas asociadas con el número de operaciones por clase. Contrasta este nivel de abstracción con las métricas que se centran en contar condiciones booleanas (complejidad ciclomática) o en contar líneas de código.

24.2.3. Ocultación de información

La ocultación de información suprime (u oculta) los detalles operacionales de un componente de programa. Solo se proporciona la información necesaria para acceder al componente a aquellos otros componentes que deseen acceder.

Un sistema OO bien diseñado debe implementar ocultación de información. Por esta razón, las métricas que proporcionan una indicación del grado de ocultación logrado suministran un'indicio de la calidad del diseño OO.

24.2.4. Herencia

La herencia es un mecanismo que habilita las responsabilidades de un objeto, para propagarse a otros objetos. La herencia ocurre a través de todos los niveles de una jerarquía de clases. En general, el software convencional no cumple esta característica.

Ya que la herencia es una característica vital en muchos sistemas **OO,** muchos métodos OO se centran en ella. Los ejemplos (discutidos más adelante en este capítulo) incluyen múltiples hijos (instancias inmediatas de una clase), múltiples padres (generalizaciones inmediatas), y jerarquías de clase a un nivel de anidamiento (profundidad de una clase en la jerarquía de herencia).

24.2.5. Abstracción

La abstracción es un mecanismo que permite al diseñador concentrarse en los detalles esenciales de un componente de programa (ya sean datos o procesos), prestando poca atención a los detalles de bajo nivel. Como Berard declara: «la abstracción es un concepto relativo. A medida que se mueve a niveles más altos de abstracción, se ignoran más y más detalles, es decir, se tiene una visión más general de un concepto o elemento. **A** medida que se mueve a niveles de abstracción más bajos, se introducen más detalles, es decir, se tiene una visión más específica de un concepto o elemento.»

Ya que una clase es una abstracción, que puede visualizarse a diferentes niveles de detalle de diferentes maneras (por ejemplo, como una lista de operaciones, como una secuencia de estados, como una serie de colaboraciones), las métricas OO representan abstracciones en términos de mediciones de una clase (por ejemplo, número de instancias por clase por aplicación, número o clases parametrizadas por aplicación, y proporción de clases parametrizadas con clases no parametrizadas).

24.3 MÉTRICAS PARA EL MODELO DE DISEÑO OO

Mucho acerca del diseño orientado a objetos es subjetivo —un diseñador experimentado «sabe» como caracterizar a un sistema **OO,** para que implemente efectivamente los requerimientos del cliente—. Pero, cuando un modelo de diseño OO crece en tamaño y complejidad, una visión más objetiva de las características del diseño puede beneficiar al diseñador experimentado (que adquiere vista adicional), **y** al novato (que obtiene indicadores de calidad que de otra manera no estarían disponibles).

¿Qué características pueden medirse cuando se evalúa un diseño OO?

Como parte de un tratamiento detallado de las métricas de software para sistemas **OO,** Whitmire [WHI97] describe nueve características distintas y medibles de un diseño **OO:**

Tamano. El tamaño se define en términos de cuatro enfoques: población, volumen, longitud y funcionalidad. La población se mide haciendo un recuento de las entidades **OO,** como las clases u operaciones. Las medidas de volumen son idénticas a las de población, pero se realizan dinámicamente - e n un instante de tiempo dado–. La longitud es la medida de una cadena de elementos de diseño interconectados (por ejemplo, la profundidad de un árbol de herencia es una medida de longitud). Las métricas de *funcionalidad* proporcionan una indicación indirecta del valor entregado al cliente por una aplicación OO.

Complejidad. Así como el tamaño, existen diferentes enfoques de la complejidad del software [ZUS97]. Whitmire la define en términos de características estructurales, examinando cómo se interrelacionan las clases de un diseño OO con otras.

Acoplamiento. Las conexiones físicas entre los elementos del diseño OO (por ejemplo, el número de colaboraciones entre clases o el número de mensajes intercambiados entre objetos), representan el acoplamiento dentro de un sistema OO.

Suficiencia. Whitmire define la suficiencia como «el grado en que una abstracción posee los rasgos mínimos necesarios, o el grado en que una componente de diseño posee características en su abstracción, desde el punto de vista de la aplicación actual». Dicho de otro modo, se hace la pregunta: «¿qué propiedades tiene que poseer esta abstracción (clase) para que sea Útil?» [WHI97]. En esencia, un componente de diseño (por ejemplo, una clase) es suficiente si refleja completamente todas las propiedades del objeto dominio de la aplicación que se modela; esto es lo que significa que la abstracción (clase), posea los rasgos imprescindibles.

Cita:
Para muchas de las decisiones para las cuales he tenido que contar con el folklore y mitos pueden usarse ahora datos cuantitativos.
Scott Whitmire

Integridad. La Única diferencia entre integridad y suficiencia es «el conjunto de características, contra las que se comparan la abstracción o componente de diseño [WHI97]». La suficiencia compara la abstracción, desde el punto de vista de la aplicación en cuestión. La integridad considera muchos puntos de vista, preguntándose: «¿Qué propiedades se requieren para representar completamente el objeto dominio del problema?» Ya que los criterios de integridad consideran diferentes puntos de vista, hay una implicación indirecta, acerca del grado en que la abstracción o componente de diseño puede ser reutilizada.

Cohesión. Así como su correspondiente en el software convencional, un componente OO debe diseñarse de manera que posea todas las operaciones trabajando conjuntamente para alcanzar un propósito Único y bien definido. La cohesión de una clase se determina examinando el grado en que «el conjunto de propiedades que posee sea parte del diseño o dominio del problema» [WHI97].



Un informe técnico de la NASA, que aborda métricas de calidad para sistemas OO, puede descargarse del **satc.gsfc.nasa.gov/support/index.html**

Originalidad. Una característica similar a la simplicidad, la originalidad (aplicada tanto a operaciones como a clases) es el grado en que una operación es atómica; esto significa que la operación no puede ser construida fuera de una secuencia de otras operaciones contenidas dentro de una clase. Una clase que exhibe un alto grado de originalidad encapsula solamente las operaciones primitivas.

Similitud. Esta métrica indica el grado en que dos o más clases son similares en términos de estructura, comportamiento, función o propósito.

Volatilidad. Como se mencionó anteriormente en este libro, pueden ocurrir cambios en el diseño cuando se modifiquen los requisitos, o cuando ocurran modificaciones en otras partes de la aplicación, resultando una adaptación obligatoria del componente de diseño en cuestión. La volatilidad de un componente de diseño OO mide la probabilidad de que un cambio ocurra.

La descripción de métricas de Whitmire para estas características de diseño se encuentra fuera del ámbito de este libro.'Los lectores interesados deben consultar [WHI97], para más detalles.

En realidad, las métricas técnicas para sistemas OO pueden aplicarse no sólo al modelo de diseño, sino también al modelo de análisis. En las secciones siguientes, se exploran las métricas que proporcionan un indicador de calidad, al nivel de las clases OO y al nivel de operación. En suma, las métricas aplicables al manejo de proyectos y pruebas también se comentarán.

24.4 MÉTRICAS ORIENTADAS A CLASES

La clase es la unidad fundamental de un sistema OO. Por esta razón, las medidas y métricas para una clase individual, la jerarquía de clases y las colaboraciones de clases poseen un valor incalculable, para el ingeniero del software que ha de evaluar la calidad del diseño. En capítulos anteriores se estudió que la clase encapsula operaciones (procesamiento) y atributos (datos). **Así** mismo, la clase «padre» es de las que heredan las subclases (algunas veces llamadas hijas), sus atributos y operaciones. La clase, normalmente, colabora con otras clases. Cada una de estas características pueden usarse como base de la medición[2],

[2] Nótese que la validez de algunas métricas, discutidas en este capítulo, actualmente se debate en la literatura técnica. Aquellos que abanderan la teoría de medición requieren de un grado de formalismo que algunas de las métricas OO no proporcionan. De cualquier manera, es razonable declarar que todas las métricas proporcionan una visión útil para el ingeniero de software.

24.4.1. La serie de métricas CK

Uno de los conjuntos de métricas OO más ampliamente referenciados, ha sido el propuesto por Chidamber y Kemcrer [CHI94]. Normalmente conocidas como la ***serie de métricas*** *CK*, los autores han propuesto seis métricas basadas en clases para sistemas OO[3].

Métodos ponderados por clase (MPC). Asumen que *n* métodos de complejidad $c_1, c_2, ... c_n$ se definen para la clase **C**. La métrica de complejidad específica que se eligió (por ejemplo, complejidad ciclomática) debe normalizarse de manera que la complejidad nominal para un método toma un valor de 10.

$$MPC = \Sigma c_i$$

para cada $i = 1$ hasta *n*.

El número de métodos y su complejidad son indicadores razonables de la cantidad de esfuerzo requerido para implementar y verificar una clase. En suma, cuanto mayor sea el número de métodos, más complejo es el árbol de herencia (todas las subclases heredan métodos de sus padres). Finalmente, a medida que crece el número de métodos para una clase dada, más probable es que se vuelvan más y más específicos de la aplicación, así que se limita el potencial de reutilización. Por todas estas razones, el MPC debe mantenerse tan bajo como sea razonable.



El número de métodos y su complejidad está directamente relacionado con el esfuerzo requerido para comprobar una clase.

Aunque podría parecer relativamente claro llevar la cuenta del número de métodos en una clase, el problema es más complejo de lo que parece, Churcher y Shepperd [CHU95] discuten este aspecto cuando escriben:

> Para contar métodos, se debe contestar la pregunta fundamental: «¿pertenece un método únicamente a la clase que lo define, o pertenece a cada clase que la hereda directa o indirectamente?». Las preguntas como esta pueden parecer triviales, ya que el sistema de ejecución las resolverá finalmente. De cualquier manera, las implicaciones para las métricas deben ser significativas.
>
> Una posibilidad es restringir el contador de la clase actual ignorando los miembros heredados. La motivación para esto podría ser que los miembros heredados ya han sido contados en las clases donde fueron definidos, así que el incremento de la clase es la mejor medida de su funcionalidad, lo que refleja su razón para existir. Para entender lo que la clase lleva a cabo, la fuente de información más importante son sus propias operaciones. Si una clase no puede responder a un mensaje (por ejemplo, le falta un método correspondiente de sí), entonces pasará su mensaje a sus clases padres.
>
> En el otro extremo, el recuento podría incluir a todos aquellos métodos definidos en la clase en cuestión, junto con los métodos heredados. Este enfoque enfatiza la importancia del espacio de estados, en lugar del incremento de la clase, para la comprensión de la clase.
>
> Entre estos extremos, existe cierto número de posibilidades diferentes. Por ejemplo, una podría restringir el recuento a la clase en cuestión y los miembros heredados directamente de sus padres. Este enfoque se basa en el argumento de que la especialización de clases padres es lo más directamente relevante en el comportamiento de las clases descendientes.

Así como la mayoría de las convenciones de recuento en métricas de software, cualquiera de los enfoques resumidos con anterioridad es aceptable, siempre que el enfoque de recuento sea aplicado consistentemente al momento de recolectar métricas.

Árbol de profundidad de herencia (APH). Esta métrica se define como «la máxima longitud del nodo a la raíz del árbol» [CHI94]. Con referencia a la Figura 24.1, el valor del APH para la jerarquía de clases mostrada es de 4.

A medida que el APH crece, es posible que clases de más bajos niveles heredarán muchos métodos. Esto conlleva dificultades potenciales, cuando se intenta predecir el comportamiento de una clase. Una jerarquía de clases profunda (el APH es largo) también conduce a una complejidad de diseño mayor. Por el lado positivo, los valores APH grandes implican un gran número de métodos que se reutilizarán.

Número de descendientes (NDD). Las subclases inmediatamente subordinadas a una clase en la jerarquía de clases se denominan sus descendientes. Con referencia a la Figura 24.1, la clase C2 tiene tres descendientes —subclases C21, C22 y C23—.

A medida que el número de descendientes crece, la reutilización se incrementa, pero además es cierto que cuando el NDD crece, la abstracción representada por la clase predecesora puede diluirse. Esto significa que existe una posibilidad de que algunos descendientes no sean miembros, realmente apropiados, de la clase predecesora. A medida que el NDD crece, la cantidad de pruebas (requeridas para ejercitar cada descendiente en su contexto operativo) se incrementará también.



La herencia es una habilidad extremadamente poderosa, que puede meterlo en problemas, si no se usa con cuidado. Utilice el APH y otros métricas relacionadas para darse una lectura de la complejidad de la jerarquía de clases.

[3] Chidamber, Darcy y Kemerer usan el término *métodos* en vez de *operaciones*. Su utilización del término es reflejada en esta sección.

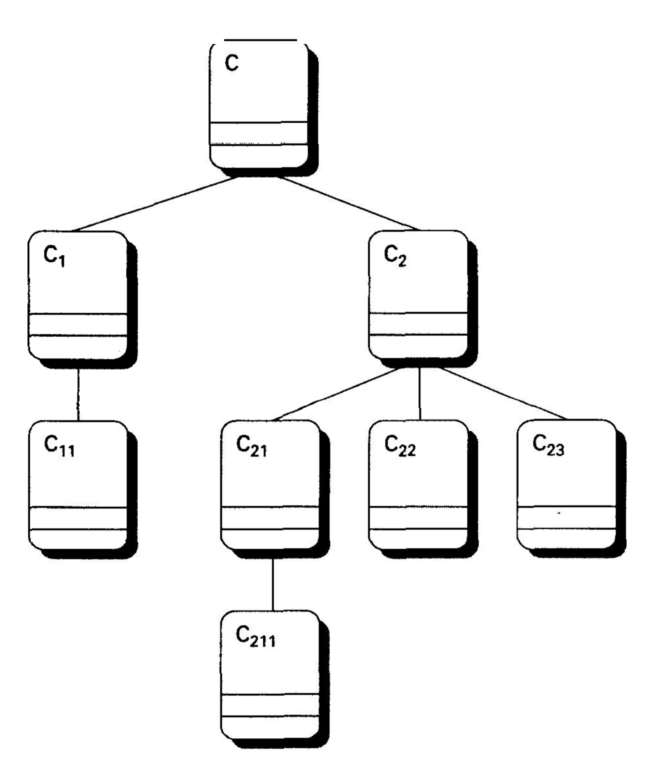


FIGURA 24.1. Una jerarquía de **clases.**

Acoplamiento entre clases objeto (ACO). El modelo CRC (Capítulo 21) debe utilizarse para determinar el valor de ACO. En esencia, ACO es el número de colaboraciones listadas para una clase, en la tarjeta índice CRC.

A medida que ACO se incrementa, es más probable que el grado de reutilización de una clase decrezca. Valores altos de ACO además complican las modificaciones y las pruebas, que se producen cuando se realizan modificaciones. En general, los valores de ACO para cada clase deben mantenerse tan bajos como sea razonable. Esto es consistente con la regla general para reducir el acoplamiento, para el software convencional.



los conceptos de acoplamiento y cohesiónse aplican tanto al software convencional como al OO: Mantenga el acoplamiento de clases bajo y la cohesión de operación alto.

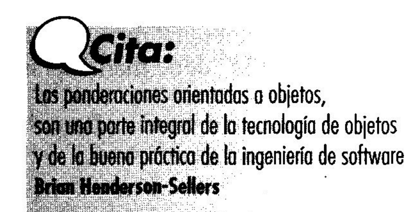
Respuesta para una clase (RPC). El conjunto de respuesta de una clase es «una serie de métodos que pueden potencialmente ser ejecutados, en respuesta a un mensaje recibido por un objeto, en la clase» [CHI94]. RPC se define como el número de métodos en el conjunto de respuesta.

A medida que la RPC aumenta, el esfuerzo requerido para la comprobación también se incrementa, ya que la secuencia de comprobación (Capítulo 23) se incrementa también. Así mismo, se dice que, así como la RPC aumenta, la complejidad del diseño global de la clase se incrementa.

Carencia de cohesión en los métodos (CCM). Cada método dentro de una clase, C, accede a uno o más atributos (también llamados variables de instancia) CCM es el número de métodos que accede a uno o más de los mismos atributos[4]. Si no existen métodos que accedan a los mismos atributos, entonces CCM = 0.

Para ilustrar el caso en el que CCM es diferente de 0, considérese una clase con 6 métodos. Cuatro de los métodos tienen uno o más atributos en común (es decir, acceden a atributos comunes). De esta manera, CCM = 4.

Si CCM es alto, los métodos deben acoplarse a otro, por medio de los atributos. Esto incrementa la complejidad del diseño de clases. En general, los valores altos para CCM implican que la clase debe diseñarse mejor descomponiendo en dos o más clases distintas. Aunque existan casos en los que un valor alto para CCM es justificable, es deseable mantener la cohesión alta, es decir, mantener CCM bajo.



24.4.2. Métricas propuestas por Lorenz y Kidd

En su libro sobre métricas **OO,** Lorenz y Kidd [LOR94] separan las métricas basadas en clases en cuatro amplias categorías: tamaño, herencia, valores internos y valores externos. Las métricas orientadas al tamaño para las clases OO se centran en el recuento de atributos y operaciones para cada clase individual, y los valores promedio **para** el sistema OO como un todo. Las métricas basadas en la herencia se centran en la forma en que las operaciones se reutilizan en la jerarquía de clases. Las métricas para valores internos de clase examinan la cohesión (Sección 24.4.1) y los aspectos orientados al código; las métricas orientadas a valores externos, examinan el acoplamiento y la reutilización. A continuación, una muestra de métricas propuestas por Lorenz y Kidd[5]:

Tamano de clase (TC). El tamaño general de una clase puede medirse determinando las siguientes medidas:

- el total de operaciones (operaciones tanto heredadas como privadas de la instancia), que se encapsulan dentro de la clase.
- el número de atributos (atributos tanto heredados como privados de la instancia), encapsulados por la clase.

[4] **La** definición formal es un poco más compleja. Véase [CHI94] para más detalle.

[5] Para un tratamiento más completo, véase [LOR94].

CONSEJO

Durante la revisión del modelo de AOO, las tarjetas CRC proporcionan una indicación razonable de las valores esperados paro TC. Si encuentra uno clase con demasiada responsobilidod durante AOO, considere el particionarla.

La métrica MPC propuesta por Chidamber y Kemerer (Sección 24.4.1) es también una métrica ponderada del tamaño de clase.

Como se indicó con anterioridad, valores grandes para TC indican que la clase debe tener bastante responsabilidad. Esto reducirá la reutilización de la clase y complicará la implementación y las pruebas. En general, operaciones y atributos heredados o públicos deben ser ponderados con mayor importancia, cuando se determina el tamaño de clase. [LOR94] Operaciones y atributos privados, permiten la especialización y son más propios del diseño.

También se pueden calcular los promedios para el número de atributos y operaciones de clase. Cuanto menor sea el valor promedio para el tamaño, será más posible que las clases dentro del sistema puedan reutilizarse.

Número de operaciones redefinidas para una subclase (NOR). Existen casos en que una subclase reemplaza una operación heredada de su superclase por una versión especializada para su propio uso. A esto se le llama *redefinición*. Los valores grandes para el NOR, generalmente indican un problema de diseño. Tal como indican Lorenz y Kidd:

> Dado que una subclase debe ser la especialización de sus superclases, deben, sobre todo, extender los servicios (operaciones) de las superclases. Esto debe resultar en nuevos nombres de métodos únicos.

Si el NOR es grande, el diseñador ha violado la abstracción representada por la superclase. Esto provoca una débil jerarquía de clases y un software OO, que puede ser difícil de probar y modificar.

Número de operaciones añadidas por una subclase (NOA). Las subclases se especializan añadiendo operaciones y atributos privados. A medida que el valor NOA se incrementa, la subclase se aleja de la abstracción representada por la superclase. En general, a medida que la profundidad de la jerarquía de clases incrementa (APH se vuelve grande), el valor para NOA a niveles más bajos en la jerarquía debería disminuir.

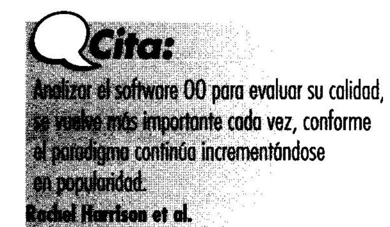
Índice de especialización (IES). El índice de especialización proporciona una indicación aproximada del grado de especialización, para cada una de las subclases en un sistema OO. La especialización se puede alcanzar añadiendo o eliminando operaciones, pero también redefiniendo.

$$IE = [NOR \times nivel] / M_{total}$$

donde *nivel* corresponde al nivel en la jerarquía de clases en que reside la clase, y M_{total} es el número total de métodos de la clase. Cuanto más elevado sea el valor de IE, más probable será que la jerarquía de clases tenga clases que no se ajusten a la abstracción de la superclase.

24.4.3. La colección de métricas MDOO

Harrison, Counseil y Nithi [HAR98] propusieron un conjunto de métricas para el diseño orientado a objetos (MDOO), que proporcionan indicadores cuantitativos para el diseño de características OO. A continuación, una muestra de métricas MDOO:



Factor de herencia de métodos (FHM). El grado en que la arquitectura de clases de un sistema OO hace uso de la herencia tanto para métodos (operaciones) como atributos, se define como:

$$FHM = \Sigma M_i(C_i) / \Sigma M_a(C_i)$$

donde el sumatorio va desde i=1 hasta TC. TC se define como el número total de clases en la arquitectura, C_i es una clase dentro de la arquitectura, y

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

donde

$M_a(C_i)$ = al número de métodos que pueden ser invocados en relación con C_i

$M_d(C_i)$ = al número de métodos declarados en la clase C_i

$M_i(C_i)$ = al número de métodos heredados (y no redefinidos) en C_i

El valor de FHM (el factor de herencia de atributo, FHA, se define de manera análoga), proporciona una referencia sobre impacto de la herencia en software OO.

Factor de acoplamiento (FA). Con anterioridad, en este capítulo se indicó que el acoplamiento es un indicador de las conexiones entre los elementos del diseño OO. La colección de métricas MDOO define el factor de acoplamiento de la siguiente manera:

$$FA = [\Sigma_i \Sigma_j \; es_cliente\,(C_i, C_j)] / (TC^2 - TC)$$

donde los sumatorios van desde $i = 1$ hasta TC y desde $j = 1$ hasta TC. La función

Es cliente = 1, si existe una relación entre la clase cliente, C_c y la clase servidora C_s y $C_c \neq C_s$.

Es_cliente = 0, en cualquier otro caso.

A pesar de que muchos factores afectan la complejidad, comprensión y mantenimiento del software, es razonable concluir que conforme el valor del **FA** crece, la complejidad del software OO también crece, **y** la comprensión, el mantenimiento y el potencial de reutilización, pueden resentirse como resultado.

Factor de polimorfismo (FP). Harrison, Counsell y Nithi [HAR98] definen FP como «el número de métodos que redefinen métodos heredados, dividida por el máximo número de posibles situaciones polimórficas distintas...; de esta manera, el FP es una medida indirecta de la cantidad relativa de ligadura dinámica en un sistema». La colección de métricas MDOO define el FP de la siguiente manera:

$$FP = \Sigma_i M_o(C_i) \,/\, \Sigma_i [M_n(C_i) \times DC(C_i)]$$

donde los sumatorios van desde $i = 1$ hasta TC y

$$M_d(C_i) = M_n(C_i) + M_o(C_i)$$

Y

$M_n(C_i)$ = al número de métodos nuevos.
$M_o(C_i)$ = al número de métodos redefinidos.
$DC(C_i)$ = al número de descendientes (el número de clases descendientes de una clase base).

Harrison, Counsell y Nithi [HAR98] presentan un análisis detallado de FHM, FA y **FP** en conjunto con otras métricas, y examinan su validez de uso, en la evaluación de la calidad del diseño.

24.5 MÉTRICAS ORIENTADAS A OPERACIONES

Ya que la clase es la unidad dominante en los sistemas **OO,** se han propuesto menos métricas para operaciones que las relacionadas con las clases. Churcher **y** Shepperd [CHU95] discuten lo anterior cuando declaran que:

> **Resultados de estudios recientes indican que los métodos tienden a ser pequeños, tanto en términos de número de sentencias como en complejidad lógica [WIL93], sugiriendo que la estructura de conectividddd de un sistema debe ser mas importante que el contenido de los módulos individuales.**

De cualquier modo, existen algunas ideas que pueden llegar a apreciarse, examinando las características promedio para los métodos (operaciones). A continuación se resumen tres simples métricas, propuestas por Lorenz y Kidd [LOR94]:

Tamaño medio de operación (TO_{medio}). Aunque las líneas de código podrían ser usadas como un indicador para el tamaño de operación, la medida LDC adolece de todos los problemas discutidos en el Capítulo 4. Por esta razón, el número de mensajes enviados por la operación proporciona una alternativa para el tamaño de operación. A medida que el número de mensajes enviados por una sola operación se incrementan, es más probable que las responsabilidades no hayan sido correctamente asignadas dentro de la clase.

Referencia cruzada

Los métricas pueden aplicarse a nivel de componentes, pero también pueden **aplicarse** a operaciones. Véase el Capítulo 19 para más detalles.

Complejidad de operación (CO). La complejidad de una operación puede ser calculada usando cualquiera de las métricas de complejidad (Capítulo 19) propuestas para el software convencional [ZUS90]. Ya que las operaciones deben limitarse a una responsabilidad específica, el diseñador debería esforzarse por mantener la CO tan baja como sea posible.

Número de parámetros de media por operación (NP_{media}). Tan largo como sea el número de parámetros de operación, más compleja será la colaboración entre objetos. En general, NP_{media} debe mantenerse tan baja como sea posible.

24.6 MÉTRICAS PARA PRUEBAS ORIENTADAS A OBJETOS

Las métricas de diseño anotadas en las Secciones 24.4 y 24.5 proporcionan una indicación de la calidad de diseño. También proveen una indicación general de la cantidad de esfuerzo de pruebas requerido para probar un sistema OO.

Binder [BIN94] sugiere que una amplia gama de métricas de diseño tienen una influencia directa en la «comprobabilidad» de un sistema OO. Las métricas se organizan en categorías, que reflejan características de diseño importantes.

Encapsulación

Carencia de cohesión en métodos (CCM)[6]. Cuanto más alto sea el valor CCM será necesario probar más estados para asegurar que los métodos no generan efectos colaterales.

[6] **Véase la Sección** 24.4.1 **para una descripción de** CCM

Porcentaje público y protegido (PPP). Los atributos públicos que se heredan de otras clases son visibles para esas clases. Los atributos protegidos son una especialización y son privados a subclases específicas. Esta métrica indica el porcentaje de atributos de una clase que son públicos. Valores altos para PPP incrementan la probabilidad de efectos colaterales entre clases. Las pruebas deben diseñarse para asegurar que ese tipo de efectos colaterales sean descubiertos.

Acceso público a datos miembros (APD). Esta métrica indica el número de clases (o métodos) que pueden acceder a los atributos de otras clases, una violación de encapsulación. Valores altos para APD producen potencialmente efectos colaterales entre clases. Las pruebas deben diseñarse para estar seguros de que ese tipo de efectos colaterales serán descubiertos.

Herencia

Número de clases raíz (NCR). Esta métrica es un recuento de las distintas jerarquías de clases, que se describen en el modelo de diseño. Se deben desarrollar las colecciones de pruebas para cada clase raíz, y la jerarquía de clases correspondiente. A medida que el NCR se incrementa, el esfuerzo de comprobación también se incrementa.



La comprobación OO puede ser un poco compleja. las métricas pueden ayudarle a la asignación de recursos de pruebas a hilos, escenarios y grupas de clases, que son «sujetos» basados en características ponderadas. Utilícelos.

Número de Padres Directos (NPD). Cuando es utilizado en el contexto OO, el NPD es una indicación de herencia múltiple. NPD > 1 indica que la clase hereda sus atributos y operaciones de más de una clase raíz. Se debe evitar que NPD > 1 tanto como sea posible.

Número de descendientes (NDD) y árbol de profundidad de herencia (APH)[7]. Tal como se explicó en el Capítulo 23, los métodos de la superclase tendrán que ser probados nuevamente para cada subclase.

Además de las métricas anteriores, Binder [BIN94] también define métricas para la complejidad y polimorfismo de las clases. Las métricas definidas para la complejidad de clase, incluyen tres métricas CK (Sección 24.4.1): Métodos ponderados por clase (MPC), el acoplamiento entre clases de objetos (ACO) y la respuesta para una clase (RPC). En resumen, también se definen las métricas asociadas con el recuento de métodos. Las métricas asociadas con el polimorfismo se especifican en detalle. Lo mejor es dejar su descripción a Binder.

24.7 MÉTRICAS PARA PROYECTOS ORIENTADOS A OBIETOS

Como se presentó en la Parte Dos de este libro, el trabajo del jefe de proyecto es planear, coordinar, registrar y controlar un proyecto de software. En el Capítulo 20 se presentaron algunos de los aspectos especiales asociados con la gestión de proyecto para proyectos OO. Pero ¿qué hay acerca de las métricas'? ¿Existen métricas OO especializadas que puedan ser utilizadas por el jefe de proyecto para proporcionar una visión interna adicional sobre el progreso de su proyecto? [8]. La respuesta, desde luego es «sí».

La primera actividad ejecutada por el jefe de proyecto es planificar, y una de las primeras tareas de planificación es la estimación. Retomando el modelo evolutivo de procesos, la planificación se vuelve a revisar después de cada iteración del software. De este modo, la planificación (y sus estimaciones de proyecto) es revisada nuevamente después de cada iteración de AOO, DOO e incluso POO.

Uno de los aspectos clave, al que debe hacer frente un jefe de proyecto durante la planificación, es una estimación del tamaño de implementación del software. El tamaño es directamente proporcional al esfuerzo y la duración. Las siguientes métricas [LOR94] pueden proporcionar una visión sobre el tamaño del software:

Referencia cruzada

La aplicabilidad de un modelo de procesos evolutivos, llamado el modelo recursivo/paralelo, se discute en el Capítulo 20.

Número de escenario (NE). El número de escenarios o casos uso (Capítulos 11 y 21) es directamente proporcional al número de clases requeridas para cubrir los requisitos, el número de estados para cada clase, el número de métodos, atributos y colaboraciones. El NE es un importante indicador del tamaño de un programa.

Número de clases clave (NCC). Una clase clave se centra directamente en el dominio del negocio para el problema, y tendrá una menor probabilidad de ser imple-

[7] Véase la Sección 24.4.1 para una descripción de NCC y APM

[8] Una descripción interesante de la colección de métricas CK (Sección 24.4.1) para el uso en la administración de la toma de decisiones puede encontrarse en [CHI98].

mentada por medio de la reutilización[9]. Por esta razón, valores altos para NCC indican gran trabajo de desarrollo substancial. Lorenz y Kidd [LOR94] sugieren que entre el 20 y el 40 por 100 de todas las clases en un sistema OO típico corresponde a las clases clave. El resto es infraestructura de soporte (GUI, comunicaciones, bases de datos, etc.).

Número de subsistemas (NSUB). El número de subsistemas proporciona una visión sobre la asignación de recursos, la planificación (con énfasis particular en el desarrollo paralelo) y el esfuerzo de integración global.

Las métricas NE, NCC y NSUB pueden recolectarse sobre proyectos OO pasados, y están relacionados con el esfuerzo invertido en el proyecto como un todo, y en actividades de procesos individuales (por ejemplo, AOO, DOO, POO y pruebas **OO)**. Estos datos pueden también utilizarse junto con métricas de diseño discutidas con anterioridad en este capítulo, para calcular «métricas de productividad», tales como el número de clases promedio por desarrollador o promedio de métodos por persona/mes. Colectivamente, estas métricas pueden usarse para estimar el esfuerzo, duración, personal y otra información de proyecto para el proyecto actual.

RESUMEN

El software orientado a objetos es fundamentalmente diferente al software desarrollado con el uso de métodos convencionales. Es por esto que las métricas para sistemas OO se enfocan en la ponderación que puede aplicarse a las clases y a las características del diseño —localización, encapsulación, ocultamiento de información, herencia y técnicas de abstracción de objetos—, que definen a la clase como única.

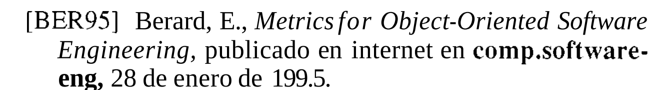
La colección de métricas CK define seis métricas de software orientadas a la clase que se centran en la clase y en la jerarquía de clases. La colección de métricas también incorpora métricas para evaluar las colaboraciones entre clases y la cohesión de métodos que residen dentro de la clase. Al nivel orientado a clases, la colección CK puede complementarse con las métricas propuestas por Lorenz y Kidd y la colección de métricas MDOO. Estas incluyen ponderaciones de «tamaño» de clase, y otras métricas que proporcionan una visión acerca del grado de especialización de las subclases.

Las métricas orientadas a operaciones se centran en el tamaño y complejidad de las operaciones individuales. Sin embargo, es importante hacer notar que la primera para las métricas de diseño OO es a nivel de clases.

Se ha propuesto una amplia variedad de métricas OO para evaluar la comprobabilidad de un sistema OO. Estas métricas se centran en la encapsulación, herencia, complejidad de las clases y polimorfismo. Muchas de estas métricas han sido adaptadas de la colección CK y de las métricas propuestas por Lorenz y Kidd. Otras han sido propuestas por Binder.

Las características ponderables del modelo de análisis y diseño pueden ayudar al jefe de proyecto de un sistema OO en la planificación y registro de las actividades. El número de escenarios (casos de uso), clases clave y subsistemas proporcionan información acerca del nivel de esfuerzo requerido para implementar el sistema.

REFERENCIAS


[BER95] Berard, E., *Metrics for Object-Oriented Software Engineering,* publicado en internet en **comp.software-eng,** 28 de enero de 199.5.

[CHI94] Chidamber, S.R., y C.F. Kemerer, «A Metrics Suite for Object-Oriented Design», *IEEE Trans. Software Engineering*, vol. 20, n.° 6, Junio de 1994, pp. 476-493.

[CHI98] Chidamber, S.R., D.P. y C.F. Kemerer, «Management Use of Metrics for Object-Oriented Software: An Exploratory Analysis», *IEEE Trans. Software Engineering,* vol. 24, n.° 8, Agosto de 1998, pp. 629-639.

[CHU95] Churcher, N.I., y **M.J.** Shepperd, «Towards a Conceptual Framework for Object-Oriented Metrics», ***ACM** Software Engineering Notes,* vol. *20*, n.° 2, Abril de 199.5, pp. 69-76.

[HAR98] Harrison, R., **S.J.** Counseil y R.V. Nithi, «An Evaluation of the MOOD Set of Object-Oriented Software Metrics», *IEEE Trans. Software Engineering*, vol. 24, n.° 6, Junio de 1998, pp. 491-496.

[LOR94] Lorenz, M., y J. Kidd, *Object-Oriented Software Metric,* Prentice-Hall, 1994.

[WHI97] Whitmire, S., *Object-Oriented Design Measuremente,* Wiley, 1997.

[ZUS90] Zuse, H., *Software Complexity: Measures and Methods,* DeGruyter, Nueva York, 1990.

[ZUS97] Zuse, H., *A framework of Software Mesurement,* DeGruyter, Nueva York, 1997.

[9] Esto sólo es verdad hasta que una robusta librería de componentes reutilizables se desarrolla para un dominio particular.

PROBLEMAS Y PUNTOS A CONSIDERAR

24.1. Revise las métricas presentadas en este capítulo y en el Capítulo 19. ¿Cómo podía caracterizar las diferencias semánticas y sintácticas entre las métricas para software convencional y OO?

24.2. ¿Cómo es que la localización afecta las métricas desarrolladas para software convencional y OO?

24.3. ¿Por qué no se hace más énfasis en las métricas OO que abordan las características específicas de las operaciones residentes dentro de una clase?

24.4. Revise las métricas descritas en este capítulo y sugiera algunas que aborden directa o indirectamente el ocultamiento de información.

24.5. Revise las métricas descritas en este capítulo y sugiera algunas que aborden directa o indirectamente la abstracción.

24.6. Una clase **x** posee 12 operaciones. Se ha calculado la complejidad ciclomática para todas las operaciones del sistema OO y el valor promedio de la complejidad del módulo es **4.** Para la clase **x** la complejidad de las operaciones de la 1 a la 12 es 5,4,3,6,8,2,2,5,5,4,4 respectivamente. Calcular MPC.

24.7. Con respecto a las Figuras 20.8a y b, calcule el valor APH para cada uno de los árboles de herencia. ¿Cuál es el valor de NDD para la clase **x2** de ambos árboles?

24.8. Acuda a [CHI94] y presente una descripción de una página referente a la definición formal de la métrica CCM.

24.9. En la Figura 20.8b, ¿cuál es el valor de NOA para las clases **x3** y **x4?**

24.10. Con respecto a la Figura 20.8b suponga que las cuatro operaciones han sido invalidadas en el árbol de herencia (jerarquía de clases), ¿cuál es el valor de IE para esa jerarquía?

24.11. Un equipo de software ha finalizado cinco proyectos hasta la fecha. Los datos siguientes han sido recogidos para todos los tamaños de los proyectos:

Número de proyecto	NGE	NCC	NSUB	Esfuerzo (días)
1	34	60	3	900
2	55	75	6	1.575
3	122	260	8	4.420
4	45	66	2	990
5	80	124	6	2.480

Se dispone de un nuevo proyecto que se encuentra en las primeras fases de **AOO.** Se estima que para este proyecto se desarrollarán 95 casos prácticos. Estimar:

a. el número total de clases que serán necesarias para implementar el sistema;

b. la cantidad total de esfuerzo que será necesaria para implementar el sistema.

24.12. Su profesor le proporciona una lista de métricas OO procedente de este capítulo. Calcule los valores de estas métricas para uno o más de los problemas que se indican a continuación:

a. el modelo de diseño para el diseño *HogarSeguro.*

b. el modelo de diseño para el sistema **SSRB** descrito en el problema 12.13.

c. el modelo de diseño para el juego de vídeo considerado en el problema 22.14.

d. el modelo de diseño para el correo electrónico considerado en el problema 22.15.

e. el modelo de diseño para el problema **CTA** considerado en el problema 22.16.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Una variedad de libros de AOO, DOO y Comprobación OO (véase *Otras lecturas y fuentes de información* en los Capítulos 20, 21 y 22) que hacen referencia de paso a las métricas OO, pero hay pocos que abordan el tema con detalle. Los libros escritos por Jacobson *(Ohject-Oriented Software Engineering,* Addison-Wesley, 1994) **y** Graham *(Ohjecto-Oriented Methods,* Addison-Wesley, segunda edición, 1993). Proporcionan un tratamiento más extenso que la mayoría.

Whitmire [WHI97] presenta el tratamiento matemático y extensamente más sofisticado de las métricas OO publicadas a la fecha. Lorenz y Kidd [LOR94] y Hendersen-Sellers *(Object-Oriented Metrics: Measures of Complexity,* Prentice-Hall, 1996) ofrecen los únicos libros dedicados a métricas OO. Otros libros dedicados a las métricas de software convencional (véase *otras lecturas y fuentes de información* en los Capítulos 4 y 19) contienen discusiones limitadas de métricas OO.

Una amplia variedad de fuentes de información para métricas orientadas a objetos y temas relacionados se encuentra disponible en intemet. Una lista reciente de referencias a sitios (páginas) web relevantes a las métricas OO pueden encontrarse en **http://www. mhhe.pressman5.com**

.

PARTE

V

TEMAS AVANZADOS EN INGENIERÍA DEL SOFTWARE

En esta parte de *Ingeniería del Software: Un enfoque práctico* se han tenido en consideración varios temas avanzados de la ingeniería del software que ayudarán a ampliar su entendimiento sobre la ingeniería del software. En los capítulos siguientes se abarcan las cuestiones siguientes:

- ¿Qué notaciones y preliminares matemáticos («métodos formales») se requieren para especificar formalmente el software?
- ¿Cuáles son las actividades técnicas clave que se llevan a cabo durante el proceso de ingeniería del software de la sala limpia?
- ¿Cómo se utiliza la ingeniería del software basada en componentes para crear sistemas a partir de componentes reutilizables?
- ¿Cuál es el impacto de la arquitectura cliente/servidor sobre la forma en que se diseña el software de comercio electrónico?
- ¿Se pueden aplicar los principios y conceptos de la ingeniería del software en productos y aplicaciones basadas en Web?
- ¿Cuáles son las actividades técnicas clave que se requieren para la ingeniería del software?
- ¿Cuáles son las opciones arquitectónicas para establecer un entorno de herramientas CASE?
- ¿Cuál es el futuro de la ingeniería del software?

Una vez que se hayan contestado estas preguntas, se comprenderán los temas que pueden tener un impacto profundo en la ingeniería del software la próxima década.

.

CAPÍTULO

25 MÉTODOS FORMALES

LOS métodos de la ingeniería del software se pueden desglosar sobre un espectro de «formalidad», que va unido ligeramente al grado de rigor matemático que se aplica durante los métodos del análisis y diseño. Por esta razón, estos métodos, descritos anteriormente dentro de este libro, se colocarían en el extremo del espectro que corresponde a lo informal. Para crear modelos de análisis y diseño, se utiliza una combinación de diagramas, texto, tablas y notaciones sencillas, aplicando sin embargo poco rigor matemático.

A continuación se considera el extremo opuesto del espectro de formalidad. Aquí se describe una especificación y un diseño utilizando una sintaxis y una semántica formal que especifican el funcionamiento y comportamiento del sistema. La especificación formal tiene una forma matemática (por ejemplo, se puede utilizar el cálculo de predicados como base para un lenguaje formal de especificación).

En una introducción sobre métodos formales, Anthony Hall [HAL90] afirma lo siguiente:

> Los métodos formales son objeto de controversia. Quienes los propugnan afirman que pueden revolucionar el desarrollo [del software]. Sus detractores piensan que resultan imposiblemente difíciles. Mientras tanto, para la mayoría de la gente los métodos formales son tan poco familiares que resulta difícil juzgar estos puntos de vista contrapuestos.

En este capítulo se exploran los métodos formales y se examina su posible impacto sobre la ingeniería del software en los años venideros.

VISTAZO RÁPIDO

¿Qué es? Estos métodos permiten al ingeniero del software crear una especificación sin ambigüedades que sea más completa y constante que las que se utilizan en los métodos convencionales u orientados a objetos. La teoría de conjuntos y las notaciones lógicas se utilizan para crear una sentencia clara de hechos (o de requisitos). Esta especificación matemática entonces se puede analizar para comprobar que sea correcta y constante. Como esta especificación se crea utilizando notaciones matemáticas. inherentemente es menos ambigua que los nodos informales de presentación.

¿Quién lo hace? Un ingeniero del software especializado crea una especificación formal.

¿Por qué es importante? En sistemas críticos para la misión y para la seguridad, un fallo puede pagarse muy caro. Cuando la computadora falla se pueden perder vidas o incluso tener graves consecuencias económicas. En dichas situaciones es esencial descubrir los errores antes de poner en operación la computadora. Los métodos formales reducen drásticamente los errores de especificación, y consecuentemente son la base del software que tiene pocos errores una vez que el cliente comienza a utilizarlo.

¿Cuáles son los pasos? El primer paso en la aplicación de los métodos formales es definir el invariante de datos, el estado y las operaciones para el funcionamiento de un sistema. El invariante de datos es una condición que se verifica mediante la ejecución de una función que contiene un conjunto de datos. Los datos almacenados forman el estado en donde una función puede acceder y alterar; y las operaciones son las acciones que tienen lugar en un sistema a medida que lee o escribe datos en un estado. Una operación se asocia a dos condiciones: una precondición y una postcondición. La notación y la heurística asociados con los conjuntos y especificaciones constructivas —operadores de conjuntos, operadores lógicos y sucesiones — forman la base de los métodos formales.

¿Cuál es el producto obtenido? Cuando se aplican métodos formales se produce una especificación representada en un lenguaje formal como Z o VDM.

¿Cómo puedo estar seguro de que lo he hecho corredamente? Debido a que los métodos formales utilizan la matemática discreta como mecanismo de especificación, para demostrar que una especificación es correcta, se pueden aplicar pruebas lógicas a cada función del sistema.

25.1 CONCEPTOS BÁSICOS

La *Encyclopedia of Software Engineering* [MAR94] define los métodos formales de la siguiente forma:

> Los métodos formales que se utilizan para desarrollar sistemas de computadoras son técnicas de base matemática para describir las propiedades del sistema. Estos métodos formales proporcionan marcos de referencia en el seno de los cuales las personas pueden especificar, desarrollar y verificar los sistemas de manera sistemática, en lugar de hacerlo ad hoc.
>
> Se dice que un método es formal si posee una base matemática estable, que normalmente vendrá dada por un lenguaje formal de especificación. Esta base proporciona una forma de definir de manera precisa nociones tales como la consistencia y completitud, y, lo que es aun más relevante, la especificación, la implementación y la corrección.



Los métodos formales tienen un potencial tremendc para mejorar la claridad y la precisión de las especificaciones de los requisitos y a la hora de encontrar tanto errores importantes como sutiles.
Steve Easterbrook et al.

Las propiedades deseadas de una especificación formal -carencia de ambigüedad, consistencia y completitud— son los objetivos de todos los métodos de especificación. Sin embargo, la utilización de métodos formales da lugar a una probabilidad mucho mayor de alcanzar estos objetivos ideales. La sintaxis formal de un lenguaje de especificación (Sección 25.4) hace posible interpretar los requisitos o el diseño de una forma única, eliminando esa ambigüedad que suele producirse cuando es cualquier lector quien interpreta un lenguaje natural (por ejemplo, el español), o una notación gráfica. Las capacidades descriptivas de la teoría de conjuntos y de la notación gráfica (Sección 25.2) hacen posible un enunciado claro de los hechos (los requisitos). Para que los hechos sean consistentes puestos de manifiesto en un lugar de una especificación, no deberán verse contradichos al establecerse en otro lugar de la misma. La consistencia se asegura mediante una demostración matemática de que los hechos iniciales se pueden hacer corresponder formalmente (mediante reglas de inferencia) con sentencias posteriores existentes dentro de la especificación.

La *completitud* es difícil de lograr, aun cuando se utilicen métodos formales. Cuando se está creando la especificación se pueden dejar sin definir algunos aspectos del sistema; quizás otras características se omitan a propósito para ofrecer a los diseñadores una cierta libertad a la hora de seleccionar un enfoque de implementación; además es imposible considerar todos los escenarios operacionales en un sistema grande y complejo. Por último, las cosas pueden omitirse simplemente por error.

Aunque el formalismo proporcionado por las matemáticas tiene un cierto atractivo para algunos ingenieros del software, hay otros (hay quien diría que la mayoría) que no consideran especialmente agradable el punto de vista matemático del desarrollo del software. Para entender por qué un enfoque formal tiene un cierto interés, es preciso considerar en primer lugar las deficiencias asociadas a los enfoques menos formales.

25.1.1. Deficiencias de los enfoques menos formales

Los métodos descritos para el análisis y diseño en las Partes Tercera y Cuarta de este libro hacen un amplio uso del lenguaje natural y de toda una gama de notaciones gráficas. Aun cuando la aplicación cuidadosa de íos métodos de análisis y diseño junto con una revisión detallada puede ciertamente llevar a un software de elevada calidad, la torpeza en la aplicación de estos métodos puede crear toda una gama de problemas. La especificación de un sistema puede contener contradicciones, ambigüedades, vaguedades, sentencias incompletas y niveles mezclados de abstracción.



Aunque un buen índice de documento no puede eliminar las Contradicciones, puede ayudar a descubrirlas. Poro las especificaciones y otros documentos hay que invertir tiempo en crear un índice completo.

Las *contradicciones* son conjuntos de sentencias que difieren entre sí. Por ejemplo, una parte de la especificación de un sistema puede afirmar que el sistema debe de monitorizar todas las temperaturas de un reactor químico mientras que otra parte, que quizás haya escrito otro miembro del personal, puede afirmar que solamente será preciso monitorizar las temperaturas que pertenezcan a un determinado intervalo. Normalmente, las contradicciones que se producen en la misma página de la especificación del sistema se pueden detectar con facilidad. Sin embargo, es frecuente que las contradicciones estén separadas por un elevado número de páginas.

Las *ambigüedades* son sentencias que se pueden interpretar de muchas maneras. Por ejemplo, el enunciado siguiente es ambiguo:

> El operador de identidad consta del nombre y la contraseña del operador; la contraseña consta de seis dígitos; debe de ser visualizada en la pantalla de seguridad, y se depositará en el archivo de registro cuando el operador se conecte con el sistema.

En este extracto, ¿La palabra «debe» alude a la contraseña o a la identidad del operador?



Cometer errores es humano, y volver a cometerlos también.
Malcolm Forbes

La *vaguedad* suele producirse porque la especificación del sistema es un documento muy voluminoso. Alcanzar un elevado nivel de precisión de forma consistente es una tarea casi imposible. Puede dar lugar a sentencias tales como «la interfaz con el sistema empleada por los operadores del radar debe ser amistosa para con el usuario», o «la interfaz virtual se basará en simples conceptos globales que sean sencillos de entender y utilizar, y, además, en escaso número». Una revisión poco detallada de estas afirmaciones podría no detectar la carencia de información útil subyacente.

La *incomplección*[1] es posiblemente uno de los problemas que se producen más frecuentemente en las especificaciones de sistemas. Por ejemplo, considérese el siguiente requisito funcional:

> El sistema debe de mantener el nivel horario del depósito a partir de los sensores de profundidad situados en el depósito. **Estos** valores deben de ser almacenados para los Últimos seis meses.

Esto describe la parte principal del almacenamiento del sistema. Supongamos que una de las órdenes del sistema es:

> La función de la orden PROMEDIO tiene que visualizar en un PC el nivel medio de agua para un sensor concreto entre dos fechas dadas.

Suponiendo que no se ofreciese más información acerca de esta orden, los detalles de la orden estarían gravemente incompletos. Por ejemplo, la descripción de la orden no incluye lo que sucedería si un usuario de sistema especificase una fecha que distase más de seis meses de la fecha actual.

La mezcla de los niveles de abstracción se produce cuando sentencias abstractas se entremezclan aleatoriamente con sentencias que se encuentran en un nivel muy inferior. Por ejemplo, sentencias tales como:

> **El** objetivo del sistema es hacer un seguimiento de stock de un almacén

pueden verse mezcladas con la siguiente:

> Cuando el encargado de **la** *carga* escribe la orden *retirar* será preciso comunicar el número de pedido, la identidad del artículo a retirar y la cantidad retirada. **El** sistema responderá con una confirmación de que la extracción es admisible.

Aun cuando dichas sentencias son importantes en la especificación de un sistema, quienes hacen la especificación suelen arreglárselas para mezclarlas de tal modo que resulta muy difícil ver toda la arquitectura funcional global del sistema.

Todos estos problema son más frecuentes que lo que uno desearía creer. Y cada uno de ellos representa una deficiencia potencial en los métodos convencionales y orientados a objetos de especificación.

25.1.2. Matemáticas en el desarrollo del software

Las matemáticas poseen muchas propiedades útiles para quienes desarrollan grandes sistemas. Una de las propiedades más útiles es que pueden describir de forma sucinta y exacta una situación física, un objeto o el resultado de una acción. La situación ideal sería que un ingeniero del software estuviera en la misma situación que un matemático dedicado a la matemática aplicada. Se debería presentar una especificación matemática de un sistema, y elaborar la solución en base a una arquitectura de software que implemente la especificación[2].

Otra de las ventajas de la utilización de las matemáticas en el proceso del software es que proporcionan una transición suave entre las actividades de ingeniería del software. En matemáticas no solo se pueden expresar especificaciones funcionales sino también diseños de sistema, y, desde luego, el código del programa es una notación matemática, aunque ciertamente sea algo verbosa.

La propiedad fundamental de las matemáticas es que admite la abstracción y es un medio excelente para el modelado. Dado que es un medio exacto, hay pocas probabilidades de ambigüedad, y las especificaciones se pueden verificar matemáticamente para descubrir contradicciones e incompletitud; y, por último, la vaguedad desaparece completamente. Además, las matemáticas se pueden utilizar para representar niveles de abstracción en la especificación de sistema de forma organizada.

Las matemáticas constituyen una herramienta ideal para el modelado. Hacen posible exhibir el esquema fundamental de la especificación y ayudan al analista y especificador del sistema a verificar una especificación para su funcionalidad, sin problemas tales como el tiempo de respuesta, las directrices de diseño, las directrices de implementación y las restricciones del proyecto que siempre estorban. También ayuda al diseñador, porque la especificación de diseño del sistema muestra las propiedades del modelo, y ofrece tan sólo los detalles suficientes para hacer posible llevar a cabo la tarea que tengamos entre manos.

Por último, las matemáticas proporcionan un elevado nivel de verificación cuando son usadas como medio de desarrollo del software. Cuando es preciso demostrar que un diseño se ajusta a una especificación y que algunos códigos de programa son el reflejo exacto de un diseño, es posible utilizar una demostración matemática. Actualmente es la práctica preferida porque no hay que hacer un gran esfuerzo para la validación inicial, y porque gran parte de la comprobación del sistema de software tiene lugar durante la prueba de aceptación y del sistema.

[1] Esta muy extentida la traducción del término *incompleteness* por *incompletitud*, pero este término en español no está aceptado por la RAE. (N. del Trad.)

[2] Una palabra de precaución es apropiada en este punto. Las especificaciones matemáticas de sistemas que se presentan en este capítulo no son tan sucintas como una expresión matemática simple. Los sistemas de software son notoriamente complejos, y no sería realista esperar que pudieran especificarse en la misma línea que las matemáticas.

25.1.3. Conceptos de los métodos formales

El objetivo de esta sección es presentar los conceptos fundamentales implicados en la especificación matemática de sistema de software, sin abrumar al lector con un excesivo nivel de detalle matemático. Para lograr esto, se utilizarán unos pocos ejemplos sencillos:

Ejemplo 1. Una tabla de símbolos. Para mantener una tabla de símbolos se utiliza un programa. Esta tabla se utiliza frecuentemente en muchos tipos distintos de aplicaciones. Consta de una colección de elementos sin duplicación. En la Figura **25.1** se muestra un ejemplo de tabla típica de símbolos en donde se representa una tabla que utiliza un sistema operativo para que contiene almacenados los nombres de los usuarios de un sistema. Otros ejemplos de tablas incluirían por ejemplo la colección de nombres del personal en un sistema de nómina, la colección de nombres de computadoras en un sistema de comunicaciones de red y la colección de destinos de un sistema que elabora horarios de trenes.

Suponga que la tabla presentada en este ejemplo no consta de más de *Maxlds* miembros de personal. Esta afirmación, que impone una restricción sobre la tabla, es un componente de una condición conocida como *invariante de datos* —una idea importante a la cual volveremos en repetidas ocasiones a lo largo del capítulo—.



Un invariante de datos es un conjunto de condiciones que son verdaderas durante la ejecución del sistema que contiene una colección de datos.

Un *invariante de datos* es una condición verdadera a lo largo de la ejecución del sistema que contiene una colección de datos. El invariante de datos, que es válido para la tabla de símbolos descrita anteriormente, posee dos componentes: (1) que la tabla no contendrá más de *Maxlds* nombres y *(2)* que no existirán nombres duplicados en la tabla. En el caso del programa de tablas de símbolos descrito anteriormente, esto significa que, independientemente del momento en que se examine la tabla de símbolos durante la ejecución del sistema, siempre contendrá un máximo de *Maxlds* identificadores de personal y no contendrá duplicados.



En los métodos formales, un «estado» es un conjunto de datos almacenados a los que el sistema accede y modifica. Una «operación» es una acción que lee o escribe datos en un estado.

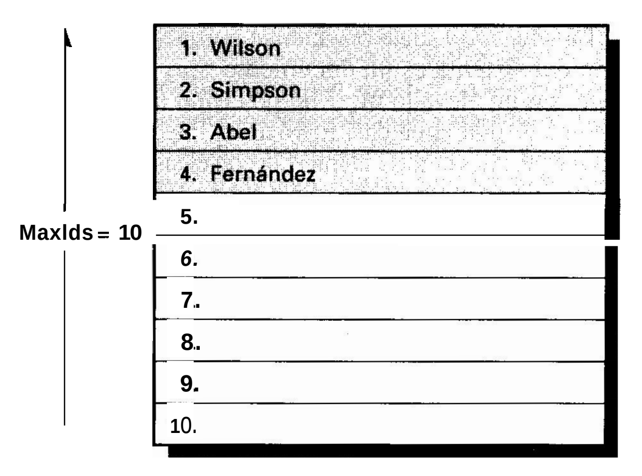


FIGURA 25.1. Una tabla de símbolos que se emplea para un sistema operativo.

Otro concepto importante es el de *estado.* En el contexto de métodos formales[3], un estado es el dato almacenado al cual accede el sistema y que es alterado por éste. En el ejemplo del programa de la tabla de símbolos, el estado es la tabla de símbolos.

El concepto final es el de *operación.* Se trata de una acción que tiene lugar en un sistema y que lee o escribe datos en un estado. Si el programa de tabla de símbolos se ocupa de añadir y eliminar nombres de personal de la tabla de símbolos, entonces estará asociado a dos operaciones: una operación para añadir un nombre especificado en la tabla de símbolos, y otra operación para eliminar un nombre existente de la tabla. Si el programa proporciona la capacidad de comprobar si existe o no un nombre concreto en la tabla, quiere decir que será necesaria una operación que proporcione alguna indicación de la existencia del nombre en la tabla.



La «precondición» define las circunstancias en las que una operación en particular es válida. La «postcondición» define que ocurre cuando una operación ha finalizado su acción.

Una operación está asociada a dos condiciones: las precondiciones y las postcondiciones. Una *precondición* define las circunstancias en que una operación en particular es válida. Por ejemplo, la precondición para una operación que añada un nombre a la tabla de símbolos de identificadores de personal es válida sólo si el nombre que hay que añadir no está almacenado en la tabla y existen menos de *Maxlds* identificadores de personal en la tabla. La *postcondición* de una operación define lo que ocurre cuando la operación ha finalizado

[3] Recuérdese que el término estado se ha utilizado también en los Capítulos 12 y 21 como una representación del comportamiento de un sistema o de objetos.

su acción. Esto se define mediante su efecto sobre el estado. En el ejemplo de la operación que añade un identificador a la tabla de símbolos de identificadores de personal, la postcondición especificaría matemáticamente que la tabla habrá sido incrementada con el identificador nuevo.

Ejemplo 2. Un gestor de bloques. Una de las partes más importantes de los sistemas operativos es el subsistema que mantiene archivos que hayan sido creados por usuarios. Una parte del subsistema de archivos es el *gestor de bloques*. Los archivos del almacén de archivos están formados por bloques de espacio que se almacenan en algún dispositivo de almacenamiento de archivos. Durante el funcionamiento de la computadora, los archivos van siendo creados y borrados, lo cual requiere la adquisición y liberación de bloques de almacenamiento. Para abordar este bloque, el subsistema de archivo mantendrá una reserva de bloques libres y seguirá también la pista de aquellos bloques que se estén utilizando en ese momento. Cuando se liberen bloques procedentes de un archivo borrado lo normal será añadirlos a una cola de bloques que están a la espera de ser añadidos al depósito de bloques que no se utilizan. En la Figura 25.2 se muestra un cierto número de componentes: la reserva de bloques no utilizados, los bloques que en la actualidad forman parte de los archivos administrados por el sistema operativo y aquellos bloques que estén esperando para ser añadidos a la reserva de espacio. Los bloques que están a la espera se mantienen en una cola y cada elemento de la cola contiene un conjunto de bloques procedentes de algún archivo borrado.



La técnica de Tormenta de ideas (Brainstorming) puede funcionar bien cuando se necesita desarrollar un invariante de datos para una función razonablemente compleja.

Para este subsistema, el estado es la colección de bloques libres, la colección de bloques usados y la cola de bloques devueltos. El invariante de datos, expresado en lenguaje natural, es:

- No habrá ningún bloque que esté a la vez usado y sin usar.
- Todos los conjuntos de bloques almacenados en la cola serán subconjuntos de la colección de bloques utilizados en ese momento.
- No existirán elementos de la cola que contengan los mismos números de bloque.
- La colección de bloques usados y bloques sin usar será la colección total de bloques de que consten los archivos.
- En la colección de bloques sin usar no existirán números de bloques duplicados.

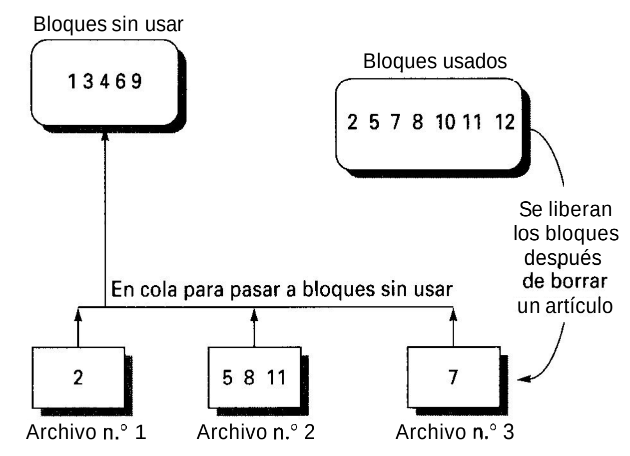


FIGURA 25.2. Un **gestor** de **bloques.**

- En la colección de bloques usados no existirán números de bloques duplicados.

Entre las operaciones asociadas a este subsistema se encuentran las siguientes:

- Una operación que añade una colección de bloques usados al final de la cola.
- Una operación que elimina una colección de bloques usados de la parte anterior de la cola y los coloca en la colección de bloques sin usar.
- Una operación que comprueba si la cola de bloques está o no vacía.

La precondición de la primera operación es que los bloques que se vayan a añadir deben de encontrarse en la colección de bloques usados. La postcondición es que la colección de bloques debe de añadirse al final de la cola.

La precondición de la segunda operación es que la cola debe de tener al menos un elemento. La postcondición es que los bloques deben de añadirse a la colección de bloques sin usar.

La operación final - comprobar si la cola de bloques proporcionados está o no vacía — no posee precondición. Esto significa que la operación siempre está definida, independientemente del valor que tenga el estado. Si la cola está vacía, la postcondición manda un valor *verdadero,* y *falso* si no lo está.

Ejemplo 3. Un concentrador de impresión. En los sistemas operativos multitarea, existe un cierto número de tareas que hacen solicitudes para imprimir archivos. Con frecuencia, no se dispone de un número suficiente de dispositivos de impresión para satisfacer simultáneamente todas las solicitudes de impresión en curso. Toda solicitud de impresión que no se pueda satisfacer de forma inmediata se ubicará en una cola a la espera de su impresión. La parte del sistema operativo que abarca la administración de estas colas se conoce con el nombre de *concentrador de impresión.*

En este ejemplo se supone que el sistema operativo no puede emplear más de *DispMax* dispositivos de

salida y que cada uno tiene asociada una cola. También se supondrá que cada uno de los dispositivos está asociado a un cierto límite de líneas por archivo que se pueden imprimir. Por ejemplo, un dispositivo de salida que tenga un límite de 1.000 líneas de impresión estará asociado a una cola que contenga tan solo aquellos archivos que no posean más de 1.000 líneas de texto. Los concentradores de impresión suelen imponer esta limitación para evitar las grandes tareas de impresión, que podrían ocupar unos dispositivos de impresión lentos durante períodos de tiempo sumamente largos. En la Figura *25.3* se muestra una representación esquemática de un concentrador de impresión.

Según se muestra en la figura, el estado del concentrador consta de cuatro componentes: las colas de archivos que esperan para ser impresos, en donde cada cola está asociada a un dispositivo de salida en particular; la colección de dispositivos controlados por el concentrador; la relación entre dispositivos de salida y el tamaño máximo de archivo que puede imprimir cada uno de ellos, y, por último, la relación entre los archivos que esperan para ser impresos y su tamaño en líneas. Por ejemplo, en la Figura 25.3 se muestra el dispositivo de salida LP1, que con **un** límite de impresión de 750 líneas, tiene dos archivos **fimp** y **personas** a la espera de imprimirse, y con un tamaño de 650 líneas y 700 líneas respectivamente.

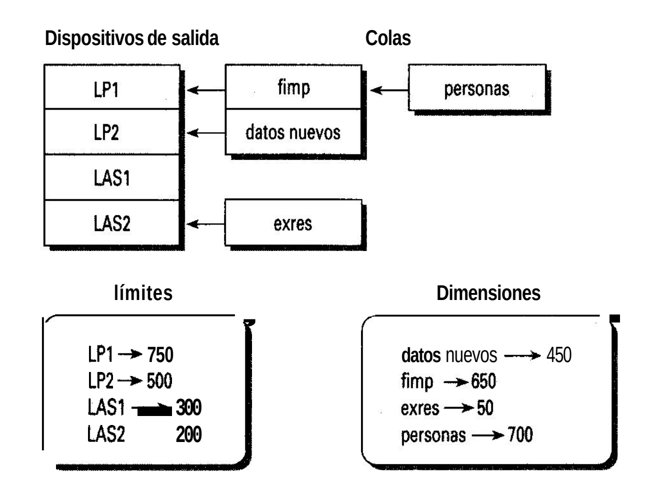


FIGURA 25.3. Un concentrador de impresión.

El estado del concentrador se representa mediante los cuatro componentes: colas, dispositivos de salida, límites y dimensiones. El invariante de datos tiene cinco componentes:

- Cada dispositivo de salida está asociado a un límite superior de líneas de impresión.
- Todo dispositivo de salida está asociado a una cola de archivos esperando impresión, que posiblemente no está vacía.
- Todo archivo tiene asociado un tamaño.
- Toda cola asociada a un dispositivo de salida contendrá archivos que tengan un tamaño menor que el límite superior de este dispositivo de salida.
- No existirán más de *DispMax* dispositivos de salida que sean administrados por este concentrador.



CLAVE

Los estados y las operaciones en muchos aspectos son análogos a la definición de clases en los sistemas orientados a objetos. Los estados representan el dominio de los datos (atributos) y los operaciones son los procesos (métodos) que manipulan los datos.

El concentrador puede tener asociado un cierto número de operaciones. Por ejemplo:

- Una operación que añada un dispositivo de salida nuevo al concentrador, junto con su límite de impresión asociado.
- Una operación que elimina un archivo de la cola asociada a un determinado dispositivo de salida.
- Una operación que añada un archivo a la cola asociada a un dispositivo de salida en particular.
- Una operación que altere el límite superior de líneas de impresión para un dispositivo de salida en particular.
- Una operación que traslade un archivo de una cola asociada a un dispositivo de salida a otra cola asociada a un segundo dispositivo de salida.

Cada una de estas operaciones se corresponde con una función del concentrador. Por ejemplo, la primera operación se correspondería con la notificación al concentrador de un nuevo dispositivo conectado al ordenador que contiene el sistema operativo que a su vez administra al concentrador.

Tal como sucedía antes, cada operación está asociada a una precondición y a una postcondición. Por ejemplo, la precondición para la primera operación es que el nombre del dispositivo de salida ya no existe y que existan en ese momento menos de *DispMax* dispositivos de salida a efectos del concentrador. La postcondición **es** que el nombre del dispositivo nuevo se añade a la colección de nombres de dispositivos ya existentes, formándose una nueva entrada para el dispositivo sin que esta entrada tenga asociados archivos en su cola, y el dispositivo se asocia a su límite de impresión.

La precondición de la segunda operación (la eliminación de un archivo de una cola asociada a un determinado dispositivo de salida) es que el dispositivo sea conocido para el concentrador y que exista al menos una entrada en la cola asociada al dispositivo. La postcondición es que la cabeza de la cola asociada al dispositivo de salida sea eliminada y se borre su entrada en la parte del concentrador que siga la pista de los tamaños de archivos.

La precondición de la quinta operación descrita anteriormente (el traslado de un archivo de una cola asociada a un dispositivo de salida a la cola asociada a un segundo dispositivo de salida) es:

- El primer dispositivo de salida es conocido para el concentrador.
- El segundo dispositivo de salida es conocido para el concentrador.
- La cola asociada al primer dispositivo contiene el archivo que hay que trasladar.
- El tamaño del archivo es menor o igual que el límite de impresión asociado al segundo dispositivo de salida.

La postcondición es que el archivo será eliminado de una cola y añadido a otra cola.

En cada uno de los ejemplos indicados en esta sección, se han presentado los conceptos clave de la especificación formal. Se ha hecho esto sin hacer hincapié en las matemáticas necesarias para hacer formal la especificación. Consideraremos estas matemáticas en la sección siguiente.

25.2 PRELIMINARES MATEMÁTICOS

Para aplicar de forma eficiente los métodos formales, el ingeniero del software debe de tener un conocimiento razonable de la notación matemática asociada a los conjuntos y a las sucesiones, y a la notación lógica que se emplea en el cálculo de predicados. El objetivo de esta sección es proporcionar una breve introducción al tema. Para una descripción más detallada del tema se recomienda examinar los libros especializados en esta materia: por ejemplo, [WIL87], [GRI93] y [ROS95].

25.2.1. Conjuntos y especificación constructiva

Un conjunto es una colección de objetos o elementos que se utiliza como la piedra angular de los métodos formales. Los elementos de un conjunto son únicos (es decir, no se permiten los duplicados). Forman un grupo pequeño de elementos dentro de llaves, separando mediante comas sus elementos. Por ejemplo, el conjunto

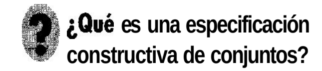
{C++, Pascal, Ada, Cobol, Java]

contiene los nombres de cinco lenguajes de programación.

El orden en que aparecen los elementos dentro de un conjunto no es relevante. El número de elementos del conjunto se conoce con el nombre de *cardinalidad*. El operador # proporciona la cardinalidad de un conjunto. Por ejemplo, la expresión

$$\#(A, B, C, D\} = 4$$

indica que se ha aplicado el operador de cardinalidad al conjunto mostrado, con un resultado que indica el número de elementos que había en el conjunto.



Hay dos maneras de definir un conjunto. En primer lugar, se puede definir por enumeración de sus elementos (esta es la forma en que se han definido los conjuntos indicados anteriormente). El segundo método consiste en crear una *especificación constructiva de conjuntos*. La forma general de los números de un conjunto se especifican empleando una expresión Booleana. Las especificaciones constructivas de conjuntos resultan preferibles a las especificaciones enumeradas, porque hacen posible definir de forma sucinta los conjuntos formados por muchos miembros. También se define explícitamente la regla que se utiliza para construir el conjunto.

Considere el siguiente ejemplo de especificación constructiva:

$$\{n : \mathbb{N} \mid n < 3 \cdot n \},$$

Esta especificación posee tres componentes: una signatura, $n : \mathbb{N}$, un predicado $n < 3$; y un término, n. La *signatura* especifica el intervalo de valores que se considerará cuando se forme el conjunto, el *predicado* (una expresión Booleana) define la forma en que se debe de construir el conjunto y, por último, el *término* ofrece la forma general del elemento del conjunto. En el ejemplo anterior, $\mathbb{N}$ denota el conjunto de los números naturales; por tanto, es preciso considerar los números naturales, el predicado indica que solamente tienen que incluir los números naturales menores que **3**, y el término especifica que todos los elementos del conjunto será de la forma n. Por tanto, la especificación anterior define un conjunto:

$$\{0, 1, 2\}$$

Cuando la forma de los elementos del conjunto es evidente, se puede omitir el término. Por ejemplo, el conjunto anterior se podría especificar en la forma:

$$\{n : \mathbb{N} \mid n < 3 \}$$

Los conjuntos que se han descrito anteriormente poseían todos ellos elementos que tienen objetos individuales. También se pueden construir conjuntos formados a partir de elementos que sean pares, ternas, etc. Por ejemplo, la especificación de conjunto

$$\{x, y : \mathbb{N} \mid x + y = 10 \cdot (x, y^2) \}$$

define el conjunto de parejas de números naturales con la forma (x, y^2) y en los cuales la suma de x e y es 10. Se trata del conjunto

$$\{(1, 81), (2, 64), (3, 49), \dots\}$$

Evidentemente, una especificación constructiva de conjuntos tal como la que se requiere para representar algunos componentes del software de computadoras puede ser considerablemente más complicada que los indicados anteriormente. Sin embargo, la misma forma y la estructura básica seguirán siendo las mismas.

25.2.2. Operadores de conjuntos

Para representar las operaciones conjuntos y las operaciones lógicas se utiliza el mismo conjunto especializado de símbolos. El ingeniero del software que pretenda aplicar los métodos formales debe de entender estos símbolos.



Cuando se desarrollan especificaciones formales es indispensable el conocimiento del conjunto de operaciones. Si se pretende aplicar métodos formales lo mejor es dedicar tiempo poro familiarizarnos con los conjuntos de operaciones.

El operador E se utiliza para indicar la pertenencia de un conjunto. Por ejemplo la expresión

x E X

tiene el valor verdadero si x es miembro del conjunto *X* y el valor *falso* en caso contrario. Por ejemplo, el predicado

12 E (6, 1, 12, 22}

tiene el valor *verdadero* por cuanto 12 es un miembro del conjunto.

El contrario del operador E es el operador $\notin$. La expresión

$$x \notin X$$

posee el valor verdadero si x no es miembro de conjunto *X* y falso en caso contrario. Por ejemplo, el predicado

$$13 \notin \{13, 1, 124, 221\}$$

posee el valor *falso*.

Los operadores $\subset$ y $\subseteq$ tienen conjuntos como operandos. El predicado

$$A \subset B$$

tiene el valor verdadero si los miembros del conjunto A están dentro del conjunto *B*, y tiene el valor *falso* en caso contrario. De esta forma el predicado

$$\{1, 2\} \subset \{4, 3, 1, 2\}$$

posee el valor verdadero. Sin embargo, el predicado

$$\{HD1, LP4, RC5\} \subset \{HD1, RC2, HD3, LP1, LP4, LP6\}$$

posee el valorfalso porque el elemento RCS no está dentro del conjunto que se encuentra a la derecha del operador.

El operador $\subseteq$ es similar a $\subset$. Sin embargo, si sus operandos son iguales posee el valor *verdadero*. Consiguientemente, el valor del predicado

$$\{HD1, LP4, RC5\} \subseteq \{HD1, RC2, HD3, LP1, LP4, LP6\}$$

es *falso*, y el predicado

$$\{HD1, LP4, RC5\} \subseteq \{HD1, LP4, RC5\}$$

es verdadero.

Un conjunto especial es el conjunto vacío O. En matemáticas normales este operador se corresponde con el cero. El conjunto vacío tiene la propiedad de que es un subconjunto de todos los conjuntos restantes. Dos identidades útiles relacionadas con el conjunto vacío son

$$\varnothing \cup A = A \text{ y } \varnothing \cap A = \varnothing$$

para todo conjunto A, en donde u es el operador unión, también conocido como «taza» (dado su forma); y $\cap$ es el operador intersección, conocido también como «gorra».

El operador unión admite dos conjuntos y forma uno con todos los elementos del conjunto después de eliminar los duplicados. Así pues, el resultado de la expresión

{Archivol, Archivo2, Impuesto, Compilador} u {NuevoImpuesto, D2, D3, Archivo2}

es el conjunto

{Archivo1, Archivo2, Impuesto, Compilador, NuevoImpuesto, D2, D3}

El operador de intersección admite dos conjuntos y forma un conjunto que consta de los elementos comunes a los dos anteriores. Por tanto, la expresión

{12, 4, 99, 1} n {1, 13, 12,771

da lugar a un conjunto {12, 1).



Las estructuras matemáticas están entre los mejores descubrimientos realizados por la mente humana.
Douglas Hofstadter

El operador diferencia de conjuntos \, como el mismo nombre indica, forma un conjunto eliminando los elementos del segundo operando de entre los elementos del primer operando. Consiguientemente el valor de la expresión

{Nuevo, Viejo, ArchivoImpuesto, ParamSistema]\{ Viejo, ParamSistema}

da lugar al conjunto {Nuevo,ArchivoImpuesto}.

El valor de la expresión

$$\{a, b, c, d\} \cap \{x, y\}$$

será el conjunto vacío O. El operador siempre proporciona un conjunto; sin embargo, en este caso no existen elementos comunes entre sus operandos, de manera que el conjunto resultante carecerá de elementos.

El operador final es el *producto cruzado* x, conocido algunas veces también como *producto cartesiano,* el cual posee dos operandos que son conjuntos de parejas. El resultado es un conjunto de parejas en donde cada una se compone de un elemento tomado del primer operando, combinado a su vez con un elemento del segundo operando. La siguiente expresión es un ejemplo del producto cruzado

$$\{1,2\} \times \{4,5,6\}$$

El resultado de esta expresión es

$$\{(1,4), (1,5), (1,6), (2,4), (2,5), (2,6)\}$$

Hay que tener en cuenta que todos los elementos del primer operando se combinan con todos los elementos del segundo.

Un concepto importante en los métodos formales es el operador *conjunto potencia.* Se trata de la colección de subconjuntos de ese conjunto. El símbolo que se utiliza para este operador de conjunto en este capítulo es ℙ. Este es un operador unario que cuando se aplica a un conjunto devuelve el conjunto de subconjuntos del operando. Por ejemplo,

$$\mathbb{P}\ \{1, 2, 3\} = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$$

ya que todos los conjuntos son subconjuntos de $\{1,2,3\}$.

25.2.3. Operadores lógicos

Otro componente importante de los métodos formales es la lógica: el álgebra de las expresiones verdaderas y falsas. Cualquier ingeniero del software entenderá el significado de los operadores lógicos comunes. Sin embargo, los operadores lógicos que están asociados a los lenguajes de programación comunes se escriben utilizando los símbolos disponibles en el teclado. Los operadores matemáticos equivalentes son los siguientes:

$\wedge$	Y
$\vee$	O
$\neg$	no
$\Rightarrow$	implica

La *cuantificación universal* es una manera de confeccionar una afirmación acerca de los elementos del conjunto que resulta ser verdadera para todos los miembros de ese conjunto. La cuantificación universal utiliza el símbolo ∀. Veamos un ejemplo de la utilización de este símbolo

$$\forall\ i, j : \mathbb{N} \cdot i > j \Rightarrow i^2 > j^2$$

en donde se lee que toda pareja de valores del conjunto de los números naturales, si i es mayor que j, entonces i^2 es mayor que j^2.

25.2.4. Sucesiones

Una sucesión es una estructura matemática que modela el hecho consistente en que sus elementos **están** ordenados. Una sucesión s es un conjunto de parejas cuyos elementos oscilan entre *1* y el elemento de mayor número. Por ejemplo,

$$\{(1, \text{Jones}), (2, \text{Wilson}), (3, \text{Shapiro}), (4, \text{Estévez})\}$$

es una sucesión. Los objetos que forman los primeros elementos de las parejas se conocen como *dominio* de una sucesión, y la colección de segundos elementos como *intervalo* de la sucesión. En este libro, las secuencias se indicarán empleando corchetes angulares. Por ejemplo, la sucesión anterior se escribiría entonces como

$$\langle\text{Jones, Wilson, Shapiro, Estévez}\rangle$$

A diferencia de los conjuntos, en una sucesión se permite la duplicidad, aunque su orden es muy importante. Por tanto

$$\langle\text{Jones, Wilson, Shapiro}\rangle \neq \langle\text{Jones, Shapiro, Wilson}\rangle$$

Una sucesión vacía se representa mediante la forma $\langle\ \rangle$.

En las especificaciones formales se utiliza un cierto número de operadores de sucesión. La concatenación, ^, es un operador binario que forma una sucesión que se construye añadiendo el segundo operando al final de su primer operando. Por ejemplo,

$$\langle 2, 3, 34, 1\rangle \,\hat{}\, \langle 12, 33, 34, 200\rangle$$

da como resultado la sucesión $\langle 2, 3, 34, 1, 12, 33, 34, 200\rangle$.

Otros operadores que pueden aplicarse a las sucesiones son *cabeza, cola, frente* y *último.* El operador *cabeza* extrae el primer elemento de una sucesión; el operador *cola* proporciona los últimos n-1 elementos finales de una sucesión de longitud n; por Último el operador *Último* extrae el elemento final de una sucesión; y el operador *frente* proporciona los últimos n-1 elementos en una sucesión de longitud n. Por ejemplo,

$$cabeza\langle 2, 3, 34, 1, 99, 101\rangle = 2$$
$$cola\langle 2, 3, 34, 1, 99, 101\rangle = \langle 3, 34, 1, 99, 101\rangle$$
$$último\langle 2, 3, 34, 1, 99, 101\rangle = 101$$
$$frente\langle 2, 3, 34, 1, 101\rangle = \langle 2, 3, 34, 1, 99\rangle$$

Dado que una sucesión es un conjunto de parejas, se pueden aplicar todos los operadores de conjuntos descritos en la Sección *25.2.2.* Cuando en un estado se utiliza una sucesión, se debería designar utilizando la palabra clave *seq.* Por ejemplo,

ListaArchivos: *seq* ARCHIVOS
NingúnUsuario: $\mathbb{N}$

describen un estado con dos componentes: una sucesión de archivos y un número natural.

25.3 APLICACIÓN DE LA NOTACIÓN MATEMÁTICA PARA LA ESPECIFICACIÓN FORMAL

Para ilustrar la utilización de la notación matemática en la especificación formal de un componente de software, estudiaremos de nuevo el ejemplo del *Gestor de Bloques* de la Sección 25.1.3. Para recapitular, veremos que se utiliza un componente importante del sistema operativo de una computadora que mantiene archivos creados por usuarios. El gestor de bloques mantiene una reserva de bloques sin utilizar y también sigue la pista de aquellos bloques que se estén utilizando en ese momento. Cuando los bloques proceden de un archivo borrado, lo normal es añadirlos a una cola de bloques en espera a ser añadidos a la reserva de bloques no utilizados. En la Figura 25.2[4] se muestra un esquema en donde se representa este funcionamiento.

¿Cómo se pueden representar los estados y los invariantes de datos utilizando los operadores lógicos y de conjuntos presentados anteriormente?

Existe un supuesto conjunto *BLOQUES*, que se compone de un conjunto formado por todos los números de bloque, y un conjunto *TodosLosBloques*, que es un conjunto de bloques entre 1 y *BloquesMax*. **Su** estado se modelará mediante dos conjuntos y una sucesión. Los dos conjuntos son *usados* y *libres*. Ambos contienen bloques, es decir, el conjunto *usados* contiene los que se están utilizando actualmente en los archivos, y el conjunto *libres* contiene los que están disponibles para los archivos nuevos. La sucesión contendrá los conjuntos de bloques listos para ser liberados procedentes de archivos que se han borrado.

El estado se puede describir de la siguiente forma

$usados, libres: \mathbb{P}\ BLOQUES$

$ColaBloques: seq\ \mathbb{P}\ BLOQUES$

Esta descripción de estado se asemeja a la declaración de variables de un programa. Afirma que *usados* y *libres* serán los conjuntos de bloques y que *ColaBloques* será una sucesión, cada uno de cuyos elementos será un conjunto de bloques. El invariante de datos se escribirá de la siguiente manera

$usados \cap libres = \emptyset\ \wedge$

$usados \cup libres = TodosBloques\ \wedge$

$\forall\ i: \text{dom}\ ColaBloques \cdot ColaBloques\ i \subseteq usados\ \wedge$

$\forall\ i\ j: \text{dom}\ ColaBloques\ .\ i \neq j \Rightarrow ColaBloques\ i \cap ColaBloques\ j = \emptyset$



Para encontrar más información sobre métodos formales visite la página **archive.comlab.ox.ac.uk/formal-methods.html**

Los componentes matemáticos del invariante de datos se corresponden con los cuatro componentes del lenguaje natural marcados que se han descrito anteriormente. En la primera línea del invariante de datos se establece que no habrá bloques comunes en la colección de bloques usados y en la colección de bloques libres. En la segunda línea se establece que la colección de bloques usados y de bloques libres será siempre igual a la colección completa de bloques del sistema. La tercera línea indica que el elemento *i-ésimo* de la cola de bloques será siempre un subconjunto de los bloques usados. La última línea afirma que **si** dos elementos de la cola de bloques no son iguales, entonces no existirán componentes comunes en estos dos elementos. Los dos últimos componentes en lenguaje natural del invariante de datos se implementan como consecuencia del hecho consistente en que usados y libres son conjuntos, y por tanto no contendrán duplicados.

La primera operación que se va a definir es la que elimina un elemento de la parte anterior de la cola de bloques. La precondición es que debe de existir al menos un elemento en la cola:

$\#ColaBloques > 0,$

La postcondición es que es preciso eliminar la cabeza de la cola y es preciso ubicarla en la colección de bloques libres, y es preciso ajustar la cola para mostrar esa eliminación:

$usados' = usados \setminus \text{cabeza}\ ColaBloques\ \wedge$

$libres' = libres \cup \text{cabeza}\ ColaBloques\ \wedge$

$ColaBloques' = cola\ ColaBloques$

Una convención que se utiliza en muchos métodos formales es que el valor de una variable después de una cierta operación lleva el signo prima. Por tanto, el primer componente de la expresión anterior afirma que el nuevo conjunto de bloques usados *(usados')* será igual al conjunto bloques usados anterior menos los bloques que hayan sido eliminados. El segundo componente afirma que el nuevo conjunto de bloques libres *(libres')* será el conjunto viejo de bloques libres después de añadir la cabeza de la cola de bloques. El tercer componente afirma que la cola

[4] Si no recuerda lo que ocurrió en la colección del gestor de bloques consulte la Sección 25.1.3 para revisar el invariante de datos, las precondiciones de las operaciones y las postcondiciones asociadas al gestor.

nueva de bloques será igual a la cola del viejo valor de la cola de bloques, es decir, a todos los elementos de la cola salvo el primero. Una segunda operación es la que se encarga de añadir una cola de bloques *BloquesA* a la cola de bloques. La precondición es que *BloquesA* sea en ese momento un conjunto de bloques usados:

$BloquesA \subseteq usados$

¿Cómo se pueden representar las pretonditiones y las posttonditiones?

La postcondición es que el conjunto de bloques se añade al final de la cola de bloques y el conjunto de bloques libres y usados permanece invariable:

$ColaBloques' = ColaBloques \wedge \langle ABlocks \rangle \wedge$
$usados' = usados$ **A**
$libres' = libres$

No cabe duda de que la especificación matemática de la cola de bloques es considerablemente más rigurosa que una narración en lenguaje natural o un modelo gráfico. Este rigor adicional requiere un cierto esfuerzo, pero los beneficios ganados a partir de una mejora de la consistencia y de la completitud puedan justificarse para muchos tipos de aplicaciones.

25.4 LENGUAJES FORMALES DE ESPECIFICACIÓN

Un lenguaje de especificación formal suele estar compuesto de tres componentes primarios: (1) una sintaxis que define la notación específica con la cual se representa la especificación; (2) una semántica que ayuda a definir un «universo de objetos» [WIN90] que se utilizará para describir el sistema; y (3) un conjunto de relaciones que definen las reglas que indican cuáles son los objetos que satisfacen correctamente la especificación.

El *dominio sintáctico* de un lenguaje de especificación formal suele estar basado en una sintaxis derivada de una notación estándar de la teoría de conjuntos y del cálculo de predicados. Por ejemplo, las variables tales como x, y, y z describen un conjunto de objetos que están relacionados a un problema (algunas a veces se denominan el *dominio del discurso)* y se utilizan junto con los operadores descritos en la Sección 25.2. Aunque la sintaxis suele ser simbólica, también se pueden utilizar iconos (símbolos gráficos como cuadros, flechas y círculos) si no son ambiguos.

El *dominio semántico* de un lenguaje de especificación indica la forma en que ese lenguaje representa los requisitos del sistema. Por ejemplo, un lenguaje de programación posee un conjunto de semánticas formales que hace posible que el desarrollador de software especifique algoritmos que transforman una entrada en una salida. Una gramática formal (tal como BNF) se puede utilizar para describir la sintaxis del lenguaje de programación. Sin embargo, un lenguaje de programación no es un buen lenguaje de especificación, porque solamente puede representar funciones computables. Un lenguaje de especificación deberá poseer un dominio semántico más amplio; esto es, el dominio semántico de un lenguaje de especificación debe de ser capaz de expresar ideas tales como «Para todo x de un conjunto infinito A, existe un y de un conjunto infinito B tal que la propiedad P es válida para x e y» [WIN90]. Otros lenguajes de especificación aplican una semántica que hace posible especificar el comportamiento del sistema. Por ejemplo, se puede desarrollar una sintaxis y una semántica para especificar los estados y las transiciones entre estados, los sucesos y sus efectos en las transiciones de estados, o la sincronización y la temporización.

Es posible utilizar distintas abstracciones semánticas para describir un mismo sistema de diferentes maneras. Eso se ha hecho de manera formal en los Capítulos 12 y 21. El flujo de datos y el procesamiento correspondiente se describía utilizando el diagrama de flujo de datos, y se representaba el comportamiento del sistema mediante un diagrama de transición entre estados. Se empleaba una notación análoga para describir los sistemas orientados a objetos. Es posible utilizar una notación de modelado diferente para representar el mismo sistema. La semántica de cada representación proporciona una visión complementaria del sistema. Para ilustrar este enfoque cuando se utilicen los métodos formales, supóngase que se utiliza un lenguaje de especificación formal para describir el conjunto de sucesos que dan lugar a que se produzca un cierto estado dentro de un sistema. Otra relación formal representa todas aquellas funciones que se producen dentro de un cierto estado. La intersección de estas dos relaciones proporciona una indicación de los sucesos que darán lugar que se produzcan funciones específicas.

En la actualidad se utiliza toda una gama de lenguajes formales de especificación: CSP [HIN95, HOR85], LARCH [GUT93], VDM [JON91] y Z [SPI88, SPI92] son lenguajes formales de especificación representativos que muestran las características indicadas anteriormente. En este capítulo, se utiliza el lenguaje de especificación Z a efectos de ilustración. Z está acompañado de una herramienta automatizada que almacena axiomas, reglas de inferencia y teoremas orientados a la aplicación que dan lugar a pruebas de corrección matemáticas de la especificación.

25.5 USO DEL LENGUAJE Z PARA REPRESENTAR UN COMPONENTE EJEMPLO DE SOFTWARE

Las especificaciones en Z se estructuran como un conjunto de esquemas —son estructuras parecidas a cuadros que presentan variables y que especifican la relación existente entre las variables—. Un esquema es, en esencia, una especificación formal análoga a la subrutina o el procedimiento de un lenguaje de programación. Del mismo modo que los procedimientos y las subrutinas se utilizan para estructurar un sistema, los esquemas se utilizan para estructurar una especificación formal.

En esta sección, se utiliza el lenguaje de especificación Z para modelar el ejemplo del gestor de bloques que se presentaba en la Sección 25.1.3 y se trataba posteriormente en la Sección 25.3. En la Tabla 25.1 se presenta un resumen de la notación del lenguaje **Z.** El siguiente ejemplo de esquema describe el estado del gestor de bloques y del invariante de datos:

La notación Z está basada en la teoría de conjuntos con tipos **y** en la lógica de primer orden. Z proporciona una estructura denominada ***esquema,*** para describir el estado **y** las operaciones de una especificación. Los esquemas agrupan las declaraciones de variables con una lista de predicados que limitan los posibles valores de las variables. En ***Z*** el esquema **X** se define en la forma

——————— **X** ———————

declaraciones

———————————————

predicados

———————————————

Las funciones constantes globales se definen en la forma

declaraciones

———————————————

predecibles

La declaración proporciona el tipo de la función o constante, mientras que el predicado proporciona su valor. En esta tabla solamente se presenta un conjunto abreviado de símbolos de Z.

Conjuntos:	
S : ℕ *X*	S se declara como un conjunto de *X.*
x ∈ *S*	x es miembro de *S*.
x ∉ S	x no es miembro de S
S ⊆ *T*	S es un subconjunto de *T*: Todo miembro de S está también en *T.*
S u *T*	La unión de S **y** *T*: Contiene todos **los** miembros de S o *T o* ambos.
S n *T*	La inserción de S **y** *T*: Contiene todos **los** miembros tanto de S como de *T*.
S \ *T*	La diferencia de **S y** *T*: Contiene todos los miembros de S salvo los que están también en *T*.
O	Conjunto vacío: No contiene miembros.
{x}	Conjunto unitario: Solamente contiene a x.
ℕ	El conjunto de **los** números naturales 0, **1, 2 ....**
S: 𝔽 *X*	Se declara S como un conjunto finito de *X*.
max (*S*)	El máximo del conjunto no vacío de números *S*.
Funciones:	
f:*X* ⤔ *Y*	Se declara como una inyección parcial de *X* e *Y*.
dom ***f***	El dominio de *f*: Dícese del conjunto de valores de x para los cuales está definido *f*(x).
ran ***f***	El rango de *f*: El conjunto de valores que toma *f*(x) cuando x recorre el dominio de *f*.
f ⊕ {*x* ↦ *y*}	Una función que coincide con ***f*** salvo que x se hace corresponder con **y.**
{*x*} ⩤ ***f***	Una función igual que *f*, salvo que x se ha eliminado de su dominio.
Lógica:	
P A Q	P **y** Q: Es verdadero si tanto P como Q son verdaderos.
P ⇒ Q	P implica Q: Es verdadero tanto si Q es verdadero como si **P** es falso.
θS′ = θS	Ningún componente del esquema ***S*** cambia en una operación.

TABLA 25.1. Resumen de la notación Z.

Gestionar Bloques

usados, libres: ℙ *BLOQUES*
ColaBloques: seq ℙ *BLOQUES*

---

usados ∩ *libres* = *O* A
usados ∩ *libres* = *TodosBloques*
∀ *i:* dom *ColaBloques* . *ColaBloques i* ⊆ *usados* A
∀ *i,j* : dom *ColaBloques* · *i* ≠ *j* ⇒
ColaBloques i ∩ *ColaBloques j* = ∅



Información detallada sobre el lenguaje Z en donde se incluye FAQ se puede encontrar en **archive.comlab. ox.ac.uk/z.html**

El esquema se compone de dos partes. La primera está por encima de la línea central representando las variables del estado, mientras que la que se encuentra por debajo de la línea central describe un invariante de los datos. Cuando el esquema que representa el invariante **y** el estado se utiliza en otro esquema, va precedido por el símbolo *A*. Por tanto, si se utiliza el esquema anterior en uno que describa, por ejemplo, una operación, se representaría mediante *AGestorBloques.* Como la afirmación anterior establece, los esquemas se pueden utilizar para describir operaciones. El ejemplo siguiente describe la operación que elimina un elemento de una cola de bloques:

Eliminar Bloques

AGestorBloques

#ColaBloques > 0,
usados' = *usados* \ *cabeza ColaBloques* A
libres' = *libres* ∪ *cabeza ColaBloques* A
ColaBloques' = *cola ColaBloques*

La inclusión de *AGestorBloques* da como resultado todas las variables que componen el estado disponible en el esquema *EliminarBloques,* y asegura que el invariante de datos se mantendrá antes y después de que se ejecute la operación.

La segunda operación, que añade una colección de bloques al final de la cola, se representa de la manera siguiente:

Añadir Bloques

AGestorBloques
BloquesA? : *BLOQUES*

BloquesA? ⊆ usados
ColaBloques' = *ColaBloques* (*BloquesA?*)
usados' = *usados* **A**
libres' = *libres*

Por convención en Z, toda variable que se lea y que no forme parte del estado irá terminada mediante un signo de interrogación. Consiguientemente, *BloquesA?*, que actúa como parámetro de entrada, acaba con un signo de interrogación.

25.6 MÉTODOS FORMALES BASADOS EN OBJETOS

El interés creciente en la tecnología de objetos ha supuesto que los que trabajan en el área de los métodos formales hayan comenzado a definir notaciones matemáticas que reflejan las construcciones asociadas a la orientación a objetos, llamadas clases, herencia e instanciación. Se han propuesto diferentes variantes de las notaciones existentes, principalmente las que se basan en Z y el propósito de esta sección es examinar Object Z que desarrollaron en el Centro de Verificación de Software de la Universidad de Queensland —.

Object Z es muy similar a Z en detalle. Sin embargo, difiere en lo que se refiere a la estructuración de los esquemas y a la inclusión de las funciones de herencia e instanciación.

A continuación se muestra un ejemplo de especificación en Object Z. Aquí se representa la especificación de una clase que describe una cola genérica que puede tener objetos de cualquier tipo.

Cola[T]

numObjetosMax : ℕ
numObjetosMax 1100

cola: seqT
#cola 1 *numObjetosMax*

INIT
cola = ⟨⟩

Añadir

Δ(numObjetosMax)
elemento? : T

#cola < numObjetosMax
cola' = cola < elemento? >

Extraer

Δ(numObjetosMax)
elemento! : T

---

cola ≠ < >
elemento! = cabeza cola
cola' = cola cola

Se ha definido una clase que tiene una variable de instancia *cola*, es decir, una secuencia que tiene objetos del tipo *T*, donde *T* puede ser de cualquier tipo; por ejemplo, puede ser un entero, una factura, un grupo de elementos de configuración o bloques de memoria. Debajo de la definición de cola se leen unos esquemas que definen la clase. La primera define una constante que tendrá un valor no mayor de 100. El siguiente especifica la longitud máxima de la cola y los dos esquemas siguientes *Añadir* y *Extraer* definen los procesos de añadir y extraer un elemento de la cola.

La precondición de *Añadir* especifica que cuando se añade a la cola un objeto *elemento?* no debe tener una longitud mayor a la permitida; y la postcondición especifica lo que ocurre cuando se ha finalizado la adición de elementos. La precondición de la operación *Extraer* especifica que para que esta operación se haga con éxito la cola no debe estar vacía, y la postcondición debe definir la extracción de la cabeza de la cola y su colocación en la variable *elemento!*

Esto es, por tanto, la especificación básica de la clase de un objeto en Object Z. Si quisiéramos utilizar un objeto definido por dicha clase, sería relativamente simple de hacer. Por ejemplo, supongamos que se está definiendo parte de un sistema operativo que manipulaba procesos que representan los programas en ejecución y que tomamos la decisión de que los procesos tenían que estar en dos colas, una con los procesos de alta prioridad y la otra con aquellos de baja prioridad, y donde la prioridad la utilice el planificador del sistema operativo con el propósito de decidir cual es el siguiente proceso a ejecutar. La primera sentencia de Object Z necesaria instanciará la clase de cola general en una que contenga procesos.

ColaProc == Cola[PROCESOS]
[procQ / cola, proc? / elemento? proc! / elemento!]

Se puede decir que todo lo que hace es reemplazar el tipo *T* por *PROCESOS* en la definición de la clase, la cola general *cola* por la cola de los procesos *procQ* y los parámetros generales de entrada y salida *elemento?* y *elemento!*, por aquellos parámetros de proceso *proc?*, y *proc!*. El símbolo / representa una forma de sustitución de texto.

El siguiente paso, como se muestra a continuación, es definir una clase que describe las dos colas. Esta clase utiliza las operaciones definidas en la clase *Cola*. Supongamos que las siguientes operaciones son necesarias:

- *añadirAlta*, añade un proceso a la cola de los procesos de alta prioridad.
- *añadirBaja*, añade un proceso a la cola de los procesos de baja prioridad.
- *INIT* inicializa las colas de alta y baja prioridad.
- *totalProcesos* devuelve el número total de procesos que están en las colas.

alta, baja : ColaProc

$alta \neq baja$

— *INIT* —

alta.INIT ∧ *baja.INIT*

total! : $\mathbb{N}$

total! =: #alta + #baja

La primera línea define el esquema. La segunda y tercera definen el estado y el invariante del estado. En este caso el estado se compone de dos colas de proceso bien diferenciadas: *alta* y *baja*.

A la definición del estado le sigue la definición de cuatro operaciones. *INIT* se define como la aplicación de la operación heredada *INIT* en las colas *alta* y *baja*; *añadirAlta* se define como la aplicación de la operación heredada *añadir* en el componente del estado *alta;* *añadirBaja* se define como la aplicación de la operación heredada *añadir* sobre el componente de estado *baja;* y, finalmente, la operación *totalProcesos* se define como la suma de tamaños de las colas individuales *alta* y *baja*.

Consecuentemente, esto es un ejemplo de instanciación en acción, donde los objetos definidos por un esquema Object Z se utilizan en otro esquema. Con esto se puede definir la herencia.

Para poder llevar a cabo la definición de herencia, examinemos otro ejemplo, el de una tabla de símbolos. Estas tablas se utilizan constantemente en aplicaciones que contienen los nombres de empleados en sistemas de personal, nombres de impresoras en sistemas operativos, nombres de carreteras en sistemas de navegación para vehículos, y otros. Para describir la herencia, primero definiremos una tabla genérica de símbolos, a continuación la utilizaremos y mostraremos cómo se puede heredar del esquema Z que la define. En primer lugar se deben describir informalmente las operaciones y el estado necesarios.

El estado estará compuesto de un conjunto del tipo T. Suponemos que no habrá más de 100 objetos y definiremos una serie de operaciones que actuarán sobre el estado:

- *INIT* que inicializa la tabla de símbolos para que esté vacía.
- *añadir* que añade un objeto a la tabla de símbolos.
- *extraer* que extrae un objeto de la tabla de símbolos.
- *número* que retorna con el número de objetos de la tabla de símbolos. A continuación se muestra esta definición.

TablaSímbolo[T]

numObjetosMax : $\mathbb{N}$

MaxElem ≤ 100
tabla : *T*

#*tabla* ≤ *MaxElem*

INIT

tabla = { }

Añadir

Δ(tabla)
objeto? : T

#tabla < MaxElem
tabla' = tabla ∪ {objeto?}

Extraer

Δ(tabla)
objeto? : T
objeto?∈ tabla

tabla' = tabla \ (objeto?]

número

Ξ(*tabla)*
numTablaIn! : $\mathbb{N}$

numTablaIn! = #*tabla*

Estas son operaciones muy simples que conllevan operaciones estándar de conjuntos como ∪, y que siguen el formato mostrado en el ejemplo de colas. La operación *añadir* se definirá solo si la tabla tiene espacio para el objeto que se va a añadir, y la operación *extraer* solo se define si el objeto que se va a extraer está dentro de la tabla de símbolos. La operación *número* no afecta al estado, y de aquí que incluya el elemento Ξ(*tabla*).

Si queremos instanciar la tabla como una tabla de nombres de empleados del tipo *EMPLEADO*, se requiere todo lo siguiente:

TabEmpleado == *TablaSimbolos*[*EMPLEADO*]
[*emp*/*tabla*,*emp*?/*objeto*?,*emp*!/*objeto*!]

donde la tabla se convierte en una tabla con objetos *EMPLEADO* y donde los parámetros de entrada definidos de forma genérica han sido sustituidos por parámetros específicos *emp?* y *emp!*

A continuación se muestra el esquema Object Z que describe la nueva tabla de empleados mediante la instanciación; simplemente involucra la redefinición de los operadores que se acaban de definir.

tablaEmp

e : TabEmpleados

AñadirEmp ≙ *e.añadir*
ExtraerEmp ≙ *e.extraer*
InitEmp ≙ *e.INIT*
NúmeroEmp ≙ *e.número*

Ahora supongamos que se necesita utilizar la herencia. Para mostrar este caso, supongamos una operación *encontrar* que devuelve un valor *encontrado* si se encuentra un objeto en la tabla de símbolos, y un valor *no encontrado* si no está en la tabla. Vamos a suponer también una aplicación en la que cuando se ejecuta la operación *encontrar* se suele utilizar el mismo objeto que está en la tabla. Aseguraremos que, cuando se aplica antes esta operación, antes de empezar se comprobará que éste es el objeto, lo que podría suponer una búsqueda prolongada en la tabla. Antes de definir este esquema se debe de utilizar el esquema original de tabla de símbolos para incluir esta información. En primer lugar necesitamos un miembro diferenciado de la tabla que se conoce como *FrecElem*. Este es el elemento que se utiliza con más frecuencia. La definición de la tabla nueva es:

TablaDif[T]

TablaSímbolos[T]
FrecElem : *T*

Encontrar

Δ(*FrecElem)*
aSerEncontrado? : *T*
estado! : {*encontrado*,*noencontrado*}

aSerEncontrado? = *FrecElem* ∨ *aSerEncontrado*? ∈ *tabla* ⇒
estado! = *encontrado* ∧ *FrecElem* ' = *aSerEncontrado?*
aSerEncontrado? ≠ *FrecElem* ∧ *aSerEncontrado*? ∉ *tabla* ⇒
estado! = *noEncontrado*

Aquí todas las operaciones definidas para *TablaSímbolos* se heredan sin cambios como está la variable de instancia *tabla*. La nueva operación *encontrar* utiliza una variable *FrecElem* que forma parte del nuevo esquema *TablaDif* de Object Z.

Esta operación comprueba si el parámetro de entrada de *encontrar* es el mismo que el elemento frecuente que se está recuperando varias veces o está dentro de la tabla. Si es así, entonces el estado correcto encontrado se devuelve y el elemento frecuente se actualiza. Si el elemento no es el elemento frecuente y no está dentro de la tabla entonces se devuelve el estado *noEncontrado*.

Este esquema se puede utilizar entonces dentro de una aplicación de empleados especificando:

TabEmpleadoFrec == *TablaDif*[*EMPLEADO*]
[*emps*/*tabla*,*emp*?/*objeto*?,*emp*!,*Empfrec*/*Objetofrec*]

Y

EmpleadoFrec

e : TabEmpleadoFrec
AñadirEmpFrec ≙ *e.AÑADIR*
ExtraerEmpFrec ≙ *e.EXTRAER*
InitEmpFrec ≙ *e.INIT*
NúmeroEmpFrec ≙ *e.NUMERO*

Esto es una introducción corta para mostrar la manera de empezar a utilizar las notaciones formales para la especificación de sistemas orientados a objetos. La mayor parte del trabajo que se ha llevado a cabo dentro de esta área ha empleado la noción Z y se ha intentado construir basado en las ideas de estado, precondición, postcondición e invariante de datos, que se ha descrito en la sección anterior, para implementar las funciones para la instanciación y la herencia. El trabajo en esta área está todavía a nivel de investigación con pocas aplicaciones. Sin embargo, durante el período de vida de esta edición las notaciones formales orientadas a objeto deberían experimentar el mismo nivel de penetración industrial que notaciones estándar tales como las notaciones Z.

25.7 ESPECIFICACIÓN ALGEBRAICA

Una de las características principales de las dos técnicas de especificación descritas anteriormente, Z y Z++, es el hecho de que están basadas en la noción de estado: una colección de datos que se ven afectados por operaciones definidas por expresiones escritas en el cálculo del predicado.

Una técnica alternativa se conoce como especificación algebraica. Y consiste en escribir sentencias matemáticas que narran el efecto de las operaciones admisibles en algunos datos. Esta técnica tiene la ventaja principal de apoyar el razonamiento de manera mucho más fácil que los métodos basados en el estado.

El primer paso al escribir una especificación algebraica es determinar las operaciones necesarias. Por ejemplo, podría darse el caso de que un subsistema que implemente una cola de mensajes en un sistema de comunicación necesite operaciones que extraigan un objeto del comienzo de la cola, que devuelvan el número de objetos de una cola y que comprueben si la cola está vacía.

Una vez que se han desarrollado estas operaciones, se puede especificar la relación que existe entre ellas. Por ejemplo, el especificador podría describir el hecho de que cuando se añade un elemento a la cola el número de elemento de esa cola aumenta en un elemento. Para mostrar una impresión del aspecto de una especificación algebraica reproduciremos dos especificaciones. La primera es para una cola, y la segunda para una tabla de símbolos. Asumimos que las operaciones siguientes son necesarias para la cola:

- *ColaVacía.* Devuelve un valor Booleano verdadero si la cola a la que se aplica está vacía y falso si no lo está.
- *añadirObjeto.* Añade un objeto al final de la cola.
- *extraerObjeto.* Extrae un objeto del final de la cola.
- *primero.* Devuelve el primer elemento de una cola, y esta no se ve afectada por esta operación.
- *Último.* Devuelve el último elemento de una cola, y esta no se ve afectada por esta operación.
- *estáVacía?* Devuelve un valor Booleano verdadero si la cola está vacía,' y falso si no lo está.

A continuación se muestran las primeras líneas de la especificación y las operaciones de esta cola:

Nombre: colaParcial
Clases: cola(Z)
Operadores:

colaVacía:	→ cola(Z)
añadirElemento:	Z x cola(Z) → cola(Z)
extraerElemento:	Z x cola(Z) → cola(Z)
primera:	cola(Z) → Z
última:	cola(Z) → Z
estáVacía?:	cola(Z) → Booleana

La primera línea es la que da el nombre al tipo de cola. La segunda línea afirma que la cola manejará cualquier tipo de entrada Z; por ejemplo, Z podría ser mensajes, procesos, empleados esperando entrar a un edificio o aeronaves esperando para aterrizar en un sistema de control del tráfico aéreo.

El resto de la especificación describe las signaturas de las operaciones: cuáles son los nombres y el tipo de elemento que procesan. La definición de *colaVacía* afirma que no toma argumentos, y que da una cola que está vacía; la definición de *añadirElemento* afirma que toma un objeto del tipo Z y una cola con los objetos del tipo Z **y** entonces da una cola de objetos del tipo Z, y así sucesivamente.

Esto es solo la mitad de la especificación—el resto describe la semántica de cada operación en función de la relación con otras operaciones—. **A** continuación, se muestran algunos ejemplos de este tipo de especificación.

La primera afirma que *estáVacía*? será verdadera para una cola vacía y falsa si existe al menos un objeto en la cola; cada una de estas propiedades se expresan en una sola línea

estáVacía?(colaVacía) = verdadera
estáVacía?(añadirObjeto(z,q)) = falsa

La definición de la operación *primera* es

primera(añadirElemento(z,q)) = **z**

la cual establece que *primero* volverá con el primer elemento de la cola.

Con esto se puede ver el estilo de especificación. Para concluir ofreceremos una especificación completa de una tabla de símbolos. Esta es una estructura de datos que contiene una colección de objetos sin duplicados. Estas tablas se encuentran prácticamente en todos los sistemas de computadoras: se utilizan

para almacenar nombres en sistemas de empleados, identificadores de aeronaves en sistemas de control del tráfico aéreo, programas en sistemas operativos y otros.

- *tablaVacía*. Construye una tabla de símbolos vacía.
- *añadirElemento*. Añade un símbolo a una tabla de símbolos que ya exista.
- *extraerElemento*. Extrae un símbolo de una tabla de símbolos que ya exista.
- *estáenTabla?* Será verdadero si un símbolo está en una tabla y falso si no está.
- *unir*. Une el contenido de dos tablas de símbolos.
- *común*. Toma dos tablas de símbolos y retorna con los elementos comunes de cada tabla.
- *esParteDe*. Retorna verdadero si una tabla de símbolos está en otra tabla de símbolos.
- *esIgual*. Retorna verdadero si una tabla de símbolos es igual a otra tabla de símbolos.
- *estáVacía*. Retorna verdadero si la tabla de símbolos está vacía.

A continuación se muestra la definición de las firmas de los operadores

Tabla de nombres

Clases: tablaSímbolos(Z)con =

Operadores:	
tablavacía:	→ tablaSímbolos(Z)
añadirElemento:	Z x tablaSímbolos (Z) → tablaSímbolos (Z)
extraerElemento:	Z x tablaSímbolos (Z) → tablaSímbolos (Z)
estáenTabla?:	Z x tablaSímbolos (Z) → Booleano
unir:	tablaSímbolos (Z) x tablaSímbolos (Z) → tablaSímbolos (Z)
común:	tablaSímbolos (Z) x tablaSímbolos (Z) → tablaSímbolos Z)
esParteDe?:	tablaSímbolos (Z) x tablaSímbolos (Z) → Booleano
esIgual?:	tablaSímbolos (Z) x tablaSímbolos (Z) → Booleano
estáVacía?:	tablaSímbolos (Z) → Booleano

Todas las definiciones se pueden entender al margen de la línea siguiente

Clases: tablaSímbolos (Z) con =

la cual establece que los objetos del tipo Z se asociarán con un operador de igualdad.

La definición del operador ***añadirElemento*** es

añadirElemento(s2, añadirElemento(s 1,s)) =
si s 1 = s2 entonces añadirElemento(s2,s)
o si no añadirElemento(s1,añadirElemento(s2,s))

Esto establece que si se realizan dos sumas en la tabla de símbolos y se intenta añadir el mismo elemento dos veces, solo será equivalente a la suma de uno de los dos elementos. Sin embargo, si los elementos difieren, entonces el efecto será el de añadirlos en un orden diferente.

La definición de *extraerElemento* es

extraerElemento(s 1,tablaVacía) = tablavacía
extraerElemento (sl, añadirElemento (s2,s)) =
si s 1 = s2 entonces extraerElemento **(s 1,s)**
o si no añadirElemento (s2,extraerElemento (s1,s))

Esta definición establece que si se intenta extraer un elemento de una tabla vacía se elaborará la construcción de tabla vacía, y que cuando se extrae un elemento s1 de una tabla que contiene un objeto *s*2, entonces si s1 y *s*2 son idénticos el efecto es el de extraer el objeto s1, y si no son iguales el efecto es el de dejar *s*2 y extraer el objeto s1.

La definición de *estáenTabla*? es

estáenTabla?(s1, tablavacía) = falso
estáenTabla(s1, añadirElemento(s2,s)) =
si s1 = s2 entonces es verdadero o si no está en Tabla? (s 1,s)

En esta definición se establece que un elemento no puede ser miembro de una tabla de símbolos vacía y que el resultado de aplicar *estaenTabla*? y *s1* a una tabla que esté formada de *s1* y *s2* devolverá un valor verdadero si s1 es igual a *s*2; si no son iguales hay que aplicar *estáenTabla*? a s1.

A continuación se muestra la definición de *unir:*

unir(s, tablavacía) = **s**
unir(s, añadirElemento(s1,t)) =
añadirElemento(s1,unir (**s,t**))

Aquí se establece que uniendo una tabla vacía y una tabla *s* da como resultado la construcción de una tabla vacía *s*, y que el efecto de unir dos tablas en donde una de ellas tenga el símbolo s1 es equivalente a añadir s1 al resultado de unir dos tablas.

A continuación se muestra la definición de *común:*

común(s, tablavacía) = tablavacía
común(s,añadirElemento(s1,t)) = si estáenTabla?(s1,s)
entonces añadirElemento(s1, común(s,t))
o sino común(s,t)

Esta definición establece que la tabla compuesta de los elementos comunes de la tabla vacía y cualquier otra tabla es siempre la tabla vacía. La segunda línea afirma que, si se unen dos tablas, entonces, si hay un elemento común *s*1, este se añade al conjunto común, o de lo contrario se forma el conjunto común de dos conjuntos.

La definición de *esParteDe*? es la siguiente:

esParteDe?(tablaVacía,s) = verdadero
esParteDe?(añadirElemento(s1,s)) =
si estáenTabla? (s1,t) entonces esParteDe?(s,t)
o si no es falso

Esta definición establece que la tabla de símbolos vacía siempre es parte de cualquier tabla de símbolos. La segunda línea establece que si la tabla *t* contiene un elemento en *s* entonces se aplica *esPartede*? para ver si *s* es una subtabla de *t*.

A continuación se presenta la definición de *esIgual*?:

esIgual?(s,t) = esParteDe?(s,t) ∧ esPartede?(t,s)

Esta definición establece que las tablas de símbolos son iguales si están dentro una de otra. La definición final de *estáVacía*? es así de sencilla:

estáVacía?(tablaVacía) = verdadero
estáVacía(añadirElemento(s1,t)) = falso

Aquí se establece simplemente que la tabla vacía está vacía y que una tabla que contiene por lo menos un objeto no estará vacía.

Por tanto, la descripción completa de la tabla de símbolos es la siguiente:

Tabla de nombres

Clases: tablaSímbolos(Z) con =

Operadores:

tablavacía: → tablaSímbolos(Z)
añadirTabla: Z x tablaSímbolos (Z) → tablaSímbolos (Z)
extraerElemento: Z x tablaSímbolos (Z) → tablaSímbolos (Z)
estáenTabla?: Z x tablaSímbolos (Z) → Booleana
unir: tablaSímbolos (Z) x tablaSímbolos (Z) → tablaSímbolos (Z)
común: tablaSímbolos (Z) x tablaSímbolos (Z) → tablaSímbolos (Z)
esParteDe?: Z x tablaSímbolos (Z) x tablaSímbolos (Z) → Booleana
esIgual?: Z x tablaSímbolos (Z) x tablaSímbolos (Z) → Booleana
estáVacía?: tablaSímbolos (Z) → Booleana

añadirElemento(s2,añadirElemento(s1,s)) =
si s1 = s2 entonces añadirElemento(s2,s)
o si no añadirElemento(s1, añadirElemento(s2,s))

extraerElemento(s1, tablavacía) = tablavacía
extraerElemento(s1, añadirElemento(s2,s)) =
si s1 = s2 entonces extraerElemento(s1,s)
o si no añadirElementos(s2,extraerElemento(s1,s))
estáenTabla?(s1, tablavacía) = falso
estáenTabla?(s1, añadirElemento(s2,s)) =
si s1 = s2 entonces verdadero
o si no estáenTabla?(s1,s)

unir(s, tablavacía) = s
unir(s, añadirElemento(s1,s)) =
añadirElemento(s1, unir(s,t))

común(s, tablavacía) = tablavacía
común(s, añadirElemento(s1,t)) =
si estáenTabla(s1,s)
entonces añadirElemento(s1, común(s,t))
o sino común (s,t)

esParteDe?(tablaVacía,s) = verdadero
esParteDe?(añadirElemento(s1,s),t) =
si estáenTabla?(s1,t) entonces esParte De?(s,t)
o si no falso

esIgual?(s,t) = esParteDe?(s,t) ∧ esParteDe?(t,s)

estáVacía?(tablaVacía) = verdadero

estáVacía?(añadirElemento(s1,s) = falso

Tales especificaciones son bastante difíciles de construir, y el problema principal es qúe no se sabe cuándo hay que parar de añadir sentencias que relaten operaciones y que tienden a ser mucho más prolongadas que sus equivalentes basados en el estado. Sin embargo, matemáticamente son más puras y más fáciles para llevar a cabo el razonamiento.

25.8 MÉTODOS FORMALES CONCURRENTES

Las dos secciones anteriores han descrito Z y la derivación orientada a objetos Object Z. El principal problema de estas especificaciones y de las notaciones matemáticas similares es que no tienen las funciones para especificar los sistemas concurrentes: aquellos donde se ejecutan un serie de procesos al mismo tiempo ~~y~~ frecuentemente con un grado elevado de comunicación entre estos procesos—. Aunque se han realizado muchos esfuerzos por modificar notaciones tales como Z para acompañar la concurrencia, no se ha hecho con mucho éxito ya que nunca se diseñaron realmente con esta idea en la cabeza. Donde sí se ha hecho con éxito ha sido en el desarrollo de notaciones formales de propósito especial para concurrencia, y el propósito de esta sección es describir una de las más conocidas: PSI (Procesos Secuenciales intercomunicados) [CSP (Communicating Sequential Processes)].

PSI fue desarrollado en Oxford por el científico informático inglés Tony Hoare. La idea principal que respalda esta notación es que se puede ver un sistema como un conjunto de procesos secuenciales (programas simples no concurrentes) que se ejecutan y se comunican con otros procesos de manera autónoma. Hoare diseñó PSI para describir tanto el desarrollo de estas notaciones como la comunicación que tiene lugar entre ellas. De la misma manera que desarrolló esta notación, también desarrolló una serie de leyes algebraicas que permiten llevar a cabo un razonamiento: por ejemplo, sus leyes permiten al diseñador demostrar que el proceso P, no se bloqueará esperando la acción de otro proceso P_2 que está a su vez esperando a que el proceso P, lleve a cabo alguna otra acción.

La notación PSI es muy simple en comparación con una notación Z u Object Z; depende del concepto de un suceso. Un suceso es algo que se puede observar, es atómico e instantáneo, lo que significa que un suceso, por ejemplo, no se puede interrumpir, sino que se completará, independientemente de lo que esté ocurriendo en un sistema. La colección de sucesos asociados con algún proceso *P* se conoce como el alfabeto de *P* y se escribe como αP. Ejemplos típicos de sucesos son el encendido de la válvula de un controlador industrial, la actualización de una variable global de un programa o la transferencia de datos a otra computadora. Los procesos se definen recursivamente en función de los sucesos. Por ejemplo

$$(e \rightarrow P)$$

describe el hecho de que un proceso está involucrado en el suceso *e* dentro de αP y entonces se comporta como *P*. En PSI se pueden anidar definiciones de procesos: por ejemplo, *P* se puede definir en función de otro proceso *P*, como

$$(e \rightarrow (e_a \rightarrow P_1))$$

en donde ocurre el suceso *e*, y el proceso se comporta como el proceso *P*,.

La funciones que se acaban de describir no son muy útiles, ya que no proporcionan ninguna opción, por ejemplo, para el hecho de que un proceso se pueda encargar de dos sucesos. El operador de opción | permite que las especificaciones PSI incluyan el elemento de determinismo. Por ejemplo, la siguiente especificación define un proceso *P* que permite una opción.

$$P = (e, \rightarrow P_1 | e_2 \rightarrow P_2)$$

Aquí el proceso *P* se define como un proceso capaz de encargarse de dos sucesos *e*, o *e*,. Si se da el primero, el proceso se comportará como *P*,, y si aparece el segundo, se comportará como *P*,.

En PSI existe una serie de procesos estándar. *PARAR* es el proceso que indica que el sistema ha terminado de manera anormal, en un estado no deseado como es el bloqueo. *SALTAR* es un proceso que termina satisfactoriamente. *EJECUTAR* es un proceso que puede encargarse de cualquier suceso en su alfabeto. Al igual que *PARAR* es un proceso no deseado e indica que ha habido un bloqueo.

Para poder hacemos una idea de la utilización de PSI en la especificación de procesos especificaremos algunos sistemas muy simples. El primero es un sistema que se compone de un proceso *C*. Una vez que se ha activado este proceso, la válvula de un reactor químico se cerrará y se parará.

$$\alpha C = \quad (cerrar\}$$
$$C = \quad (cerrar \rightarrow PARAR)$$

La primera línea define el alfabeto del proceso como un proceso que se compone de un suceso, y la segunda línea establece que el proceso *C* se encargará de *cerrar* y entonces parar.

Esta es una especificación muy simple y algo irreal. Definamos ahora otro proceso C_1 que abrirá la válvula, la cerrará y que comenzará otra vez a comportarse entonces como C,.

$$\alpha C_1 = \quad \{abrir, cerrar\}$$
$$C, = \quad (abrir \rightarrow cerrar \rightarrow C,)$$

Una definición alternativa donde se definan dos procesos sería

$$\alpha C_1 = \quad \{abrir\}$$
$$C_1 = \quad (abrir \rightarrow C,)$$

Y

$$\alpha C_2 = \quad (cerrar)$$
$$C, = \quad (cerrar \rightarrow C,)$$

Aquí, el primer proceso C_1 tiene el alfabeto *{abrir)*, se encarga de un solo suceso que aparece dentro del alfabeto y que se comporta entonces como el proceso C,. El C_2 también posee un alfabeto de un solo suceso, se encarga de este suceso y se comporta entonces como C,. Un observador ajeno al tema que vea el sistema expresado de esta forma vería la siguiente sucesión de sucesos:

abrir, cerrar, abrir, cerrar.. ..

Esta sucesión se conoce como *rastreo* del proceso. A continuación, se muestra otro ejemplo de especificación PSI. Esta representa un robot que se puede encargar de dos sucesos que se corresponden con un retroceso o un avance en la línea. Se puede establecer la definición del camino infinito de un robot sin puntos finales en el recorrido de la siguiente manera

$$\alpha ROBOT = \{avance, retroceso\}$$
$$ROBOT = (avance \rightarrow ROBOT | retroceso \rightarrow ROBOT)$$

Aquí el robot se puede encargar de avanzar o retroceder y comportarse como un robot, pudiendo avanzar y retroceder, y así sucesivamente. Supongamos otra vez que la especificación es real introduciendo algunas otras funciones PSI. La complicación es que la pista sobre la

que viaja el robot se compone de cientos de movimientos con avances y retrocesos que denotan este movimiento. Supongamos también que en esta pista el robot arranca de la posición 1. A continuación se muestra la definición de este *ROBOT,:*

$$
\begin{aligned}
\alpha ROBOT_c &= \{\text{avance,retroceso}\} \\
ROBOT, &= R_1 \\
R_1 &= (avance \rightarrow R,) \\
R_i &= (avance \rightarrow R_{i+i} \mid retroceso \rightarrow R,_1) \\
& \quad (i < 100 \wedge i > 1) \\
R_{100} &= (retroceso \rightarrow R_{99})
\end{aligned}
$$

Aquí el robot representa un proceso con los mismos dos sucesos del alfabeto como el robot anterior. Sin embargo, la especificación de **su** implicación en estos sucesos es más compleja. La segunda línea establece que el robot arrancará en una posición inicial y se comportará de la misma manera que con el proceso *R,* el cual representa **su** posición en el primer recuadro. La tercera línea establece que el proceso *R,* solo se puede encargar del suceso de avance ya que se encuentra en el punto final. La cuarta línea define la serie de procesos desde *R,* a R_{99}. Aquí se define el hecho de que en cualquier punto el robot se puede avanzar o retroceder. La línea final especifica lo que sucede cuando el robot ha alcanzado el final del recorrido: solo puede retroceder.

Esta es la manera en que funcionan los procesos en **PSI.** En sistemas reales existirá una serie de procesos ejecutándose concurrentemente y comunicándose con otros, como se muestra a continuación mediante algunos ejemplos:

- En un sistema cliente/servidor, un solo servidor en ejecución como proceso estará en comunicación con varios clientes que igualmente se ejecutarán como procesos.
- En sistemas de tiempo real, que controlan un reactor químico, existirán procesos de supervisión de la temperatura de los reactores que se comunicarán con los procesos que abren y cierran las válvulas de estos reactores.
- En un sistema de control de tráfico aéreo, la funcionalidad del radar se podría implementar como procesos que se comunican con procesos que llevan a cabo las tareas de visualizar en pantalla las posiciones de las aeronaves.

De aquí que exista la necesidad de representar en **PSI** la ejecución en paralelo de los procesos. También existe la necesidad de algún medio con el que se produzca la comunicación y la sincronización entre estos procesos. El operador que especifica la ejecución en paralelo en PSI es ||. Por tanto, el proceso *P* que representa la ejecución en paralelo de los procesos de *P,* y *P,* se define como

$$P = P, \| P,$$

La sincronización entre los procesos se logra mediante procesos con algún solapamiento en sus alfabetos. La norma sobre la comunicación y la sincronización es que cuando dos procesos se están ejecutando en paralelo, donde tienen algunos sucesos en común, deben de ejecutar ese suceso simultáneamente. Tomemos como ejemplo un sistema bastante simple y real que enciende y apaga las luces y que se basa en un ejemplo similar al de [HIN95]. Las definiciones de los dos procesos de este sistema son:

$$
\begin{aligned}
\alpha LUZ_1 &= \{encendida, apagada] \\
LUZ, &= (encendida \rightarrow encendida \rightarrow PARAR \\
& \quad \mid apagada \rightarrow apagada \rightarrow PARAR) \\
\alpha LUZ_2 &= \{encendida, apagada\} \\
LUZ, &= (encendida \rightarrow apagada \text{ -+} LUZ,)
\end{aligned}
$$

Ambos procesos tienen el mismo alfabeto. El primer proceso *LUZ,,* o bien enciende la luz y la vuelve a encender de nuevo (hay que recordar que es posible que otros procesos hayan apagado el sistema entre medias, y se ha averiado *(PARAR* es un estado no deseado), o bien la apaga una vez, y una vez más. El proceso *LUZ,* enciende la luz y la apaga, y a continuación empieza a comportarse de nuevo como *LUZ,.* Los dos procesos en paralelo se denotan mediante:

$$LUZ, \| LUZ,$$

¿Cuál es el efecto de ejecutar estos procesos en paralelo? En una introducción como es esta no se puede entrar en mucho detalle. Sin embargo, se puede dar una idea del razonamiento que se puede aplicar a dicha expresión. Se recordará que cuando se introdujo PSI se afirmó que no solo consiste en una notación para especificar los procesos concurrentes, sino que también contiene una serie de normas que permiten razonar a cerca de las expresiones de procesos complejos y, por ejemplo, determinar si cualquier suceso nefasto como un bloqueo aparecerá dentro del sistema especificado en PSI. Para poder ver lo que ocurre merece la pena aplicar algunas de las normas que desarrolló Hoare para PSI. Recordemos que se intenta averiguar cuál es el proceso que se ha definido mediante la ejecución en paralelo de los procesos *LUZ,* y *LUZ,.* Esta expresión es equivalente a:

$$
\begin{aligned}
LUZ_1 &= (encendido \rightarrow encendido \text{ -+} PARAR \\
& \quad \mid apagado \rightarrow apagado \rightarrow PARAR) \\
LUZ, &= (encendido \text{-+ } apagado \rightarrow LUZ_2)
\end{aligned}
$$

Una de las normas de Hoare establece que

$$(e \rightarrow P) \| (e \rightarrow Q) = e \rightarrow (P \| Q)$$

.Esto nos permite ampliar la expresión que involucra a LUZ, y LUZ, con

$$(encendido \rightarrow ((encendido \rightarrow PARAR) \| (apagado \rightarrow LUZ_2)))$$

Esta expresión se puede simplificar aun más utilizando otra de las normas de Hoare a la expresión

$$(encendido \text{ -+} PARAR)$$

lo cual significa que la ejecución en paralelo de los dos procesos es equivalente a una luz que se enciende **y** entonces se para en un estado de bloqueo no deseado.

Hasta ahora, se han visto procesos que cooperan con la sincronización mediante el hecho de que tienen alfabetos similares o que se solapan. En los sistemas reales también existirá la necesidad de que los procesos se comuniquen con datos. **PSI** es un método uniforme para la comunicación de datos: se trata simplemente como un suceso en donde el suceso *nomCanal.val* indica que ha ocurrido un suceso que se corresponde con la comunicación del valor *val* a través de un canal *nomCanal.* PSI contiene un dispositivo notacional similar al de Z para distinguir entre la entrada y la salida; para distinguir la primera se introduce el signo de interrogación, mientras que para distinguir la segunda se utiliza el signo de exclamación.

A continuación, se muestra un ejemplo basado en [HIN95], en donde se representa la especificación de un proceso que toma dos enteros y forma el producto.

$$
\begin{aligned}
PROD, &= A_{\langle\rangle} \\
&= (en?x \rightarrow A_{\langle x \rangle}) \\
A_{\langle x \rangle} &= (en?y \rightarrow A_{\langle x,y \rangle}) \\
A_{\langle x,y \rangle} &= (fuera!(x * y) \rightarrow A_{\langle y \rangle})
\end{aligned}
$$

A primera vista esto parece muy complicado, por eso merece la pena describirlo línea a línea. La primera define el proceso *PROD*, el cual equipara este proceso con **A**, sin entradas esperando. La segunda línea establece que cuando el proceso recibe un valor de entrada x se comporta como el proceso con un valor x almacenado. La tercera línea establece que cuando un proceso con un valor almacenado recibe un valor *y* entonces se comporta como un proceso con dos valores almacenados. La última línea establece que un proceso con dos valores almacenados emitirá el producto de estos valores, almacenará el último valor y entonces se comportará como **A** con ese valor almacenado, de manera que, por ejemplo, si aparece otro valor se multiplicará por ese valor y se emitirá por el canal de salida. **Así** pues, la especificación anterior se comporta como un proceso en donde se recibe una sucesión de enteros y lleva a cabo la multiplicación de los valores.

Se puede decir entonces que esta es una definición breve de PSI —al igual que todos los métodos formales incluye también una notación matemática y las normas para el razonamiento—. Aunque en las descripciones de Z y Object Z no se han examinado las normas y se ha concentrado en el formalismo todavía contienen funciones sustanciales para razonar sobre las propiedades de un sistema.

25.9 LOS DIEZ MANDAMIENTOS DE LOS MÉTODOS FORMALES

La decisión de hacer uso de los métodos formales en el mundo real no debe de adoptarse a la ligera. Bowen y Hinchley [BOW95] han acuñado «los diez mandamientos de los métodos formales», como guía para aquellos que estén a punto de embarcarse en este importante enfoque de la ingeniería del software[5].



La decisión de utilizar métodos formales no debería tomarse a lo ligera. Siga estos «mandamientos» y asegúrese de que todo el mundo haya recibido la formación adecuada.

1. **Seleccionarás la notación adecuada.** Con objeto de seleccionar eficientemente dentro de la amplia gama de lenguajes de especificación formal existente, el ingeniero del software deberá considerar el vocabulario del lenguaje, el tipo de aplicación que haya que especificar y el grado de utilización del lenguaje.

II. **Formalizarás, pero no de más.** En general, resulta necesario aplicar los métodos formales a todos los aspectos de los sistemas de cierta envergadura. Aquellos componentes que sean críticos para la seguridad serán nuestras primeras opciones, e irán seguidos por aquellos componentes cuyo fallo no se pueda admitir (por razones de negocios).

III. **Estimarás los costes.** Los métodos formales tienen unos costes de arranque considerables. El entrenamiento del personal, la adquisición de herramientas de apoyo y la utilización de asesores bajo contrato dan lugar a unos costes elevados en la primera ocasión. Estos costes deben tenerse en cuenta cuando se esté considerando el beneficio obtenido frente a esa inversión asociada a los métodos formales.

IV. **Poseerás un experto en métodos formales a tu disposición.** El entrenamiento de expertos y la asesoría continua son esenciales para el éxito cuando se utilizan los métodos formales por primera vez.

V. **No abandonarás tus métodos formales de desarrollo.** Es posible, y en muchos casos resulta deseable, integrar los métodos formales con los métodos convencionales y/o con métodos orientados a objetos (Capítulos 12 y 21). Cada uno de estos métodos posee **sus** ventajas y **sus** inconvenientes. Una combinación de ambos, aplicada de forma adecuada, puede producir excelentes resultados[6].

[5] Esta descripción es una versión sumamente abreviada de [BOW95].

[6] La ingeniería del software de la sala limpia (Capítulo *26)* es un ejemplo integrado que hace uso de los métodos formales y **de** una notación más convencional para el desarrollo..



Información útil sobre los métodos formales se puede obtener en: **www.clcam/users/mgh1001**

VI. **Documentarás suficientemente.** Los métodos formales proporcionan un método conciso, sin ambigüedades y consistente para documentar los requisitos del sistema. Sin embargo, se recomienda que se adjunte un comentario en lenguaje natural a la especificación formal, para que sirva como mecanismo para reforzar la comprensión del sistema por parte de los lectores.

VII. **No comprometerás los estándares de calidad.** «Los métodos formales **no** tienen nada de mágico» [BOW94], y, por esta razón, las demás actividades de **SQA** (Capítulo 8) deben de seguir aplicándose cuando se desarrollen sistemas.

VIII. **No serás dogmático.** El ingeniero de software debe reconocer que los métodos formales no son una garantía de corrección. Es posible (o como algunos probablemente dirían) que el sistema final, aun cuando se haya desarrollado empleando métodos formales, siga conteniendo pequeñas omisiones, errores de menor importancia y otros atributos que no satisfagan nuestras expectativas.

IX. **Comprobarás, comprobarás y volverás a comprobar.** La importancia de la comprobación del software se ha descrito en los Capítulos 17, 18 y 23. Los métodos formales no absuelven al ingeniero del software de la necesidad de llevar a cabo unas comprobaciones exhaustivas y bien planeadas.

X. **Reutilizarás cuanto puedas.** A la larga, la única forma racional de reducir los costes del software y de incrementar la calidad del software pasa por la reutilización (Capítulo 27). Los métodos formales no modifican esta realidad. De hecho, quizás suceda que los métodos formales sean un enfoque adecuado cuando es preciso crear componentes para bibliotecas reutilizables.

25.10 MÉTODOS FORMALES: EL FUTURO

Aun cuando las técnicas de especificación formal, con fundamento matemático, todavía no se utilizan con demasiada frecuencia en la industria) éstas ofrecen ciertamente unas ventajas substanciales con respecto a las técnicas menos formales. Liskov y Bersins [LIS86] resumen esto en la manera siguiente:

> Las especificaciones formales se pueden estudiar matemáticamente, mientras que las especificaciones informales no pueden estudiarse de esta manera. Por ejemplo, se puede demostrar que un programa correcto satisface sus especificaciones, o bien se puede demostrar que dos conjuntos alternativos de especificaciones **son** equivalentes... Ciertas formas de falta de completitud o de inconsistencia se pueden detectar de forma automática.

Además, la especificación formal elimina la ambigüedad, y propugna un mayor rigor en las primeras fases del proceso de ingeniería del software.

Pero siguen existiendo problemas. La especificación formal se centra fundamentalmente en las funciones y los datos. La temporización, el control y los aspectos de comportamiento del problema son más difíciles de representar. Además, existen ciertas partes del problema (por ejemplo, las interfaces hombre-máquina) que se especifican mejor empleando técnicas gráficas. Por último, la especificación mediante métodos formales es más difícil de aprender que otros métodos de análisis que se presentan en este libro y representa «un choque cultural)) significativo para algunos especialistas del software. Por esta razón, es probable que las técnicas de especificación formales matemáticas pasen a ser el fundamento de una futura generación de herramientas CASE. Cuando esto suceda, es posible que las especificaciones basadas en matemáticas sean adoptadas por un segmento más amplio de la comunidad de la ingeniería del software[7].

RESUMEN

Los métodos formales ofrecen un fundamento para entornos de especificación que dan lugar a modelos de análisis más completos, consistentes y carentes de ambigüedad, que aquellos que se producen empleando métodos convencionales u orientados a objetos. Las capacidades descriptivas de la teoría de conjuntos y de la notación lógica capacitan al ingeniero del software para crear un enunciado claro de los hechos (requisitos).

Los conceptos subyacentes que gobiernan los métodos formales son: (1) los invariantes de datos - con-

[7] Es importante tener en cuenta que hay otras personas que no están de acuerdo. Véase [YOU94].

diciones que son ciertas a lo largo de la ejecución del sistema que contiene una colección de datos—; (2) el estado —los datos almacenados a los que accede el sistema y que son alterados por él—; **(3)** la operación —una acción que tiene lugar en un sistema y que lee o escribe datos en un estado—. Una operación queda asociada con dos condiciones: una precondición y una postcondición.

La matemática discreta —la notación y práctica asociada a los conjuntos y a la especificación constructiva, a los operadores de conjuntos, a los operadores lógicos y a las sucesiones —constituyen la base de los métodos formales. Estas matemáticas están implementadas en el contexto de un lenguaje de especificación formal, tal como Z.

Z, al igual que todos los lenguajes de especificación formal, posee tanto un dominio semántico como un dominio sintáctico. El dominio sintáctico utiliza una simbología que sigue estrechamente a la notación de conjuntos y al cálculo de predicados. El dominio semántico capacita al lenguaje para expresar requisitos de forma concisa. La estructura Z contiene esquemas, estructuras en forma de cuadro que presentan las variables y que especifican las relaciones entre estas variables.

La decisión de utilizar métodos formales debe considerar los costes de arranque, así como los cambios puntuales asociados a una tecnología radicalmente distinta. En la mayoría de los casos, los métodos formales ofrecen los mayores beneficios para los sistemas de seguridad y para los sistemas críticos para los negocios.

REFERENCIAS

[BOW95] Bowan, J.P., y M.G. Hinchley, «Ten Commandments of Formal Methods, *Computer*», vol. 28, n.º 4, Abril 1995.

[GRI93] Gries, D., y F.B. Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, 1993.

[GUT93] Guttag, J.V., y J.J. Horning, *Larch: Languages and Toolsfor Formal Specifications*, Springer-Verlag, 1993.

[HAL90] Hall, A., «Seven Myths of Formal Methods», IEEE Software, Septiembre 1990.

[HOR85] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[HIN95] Hinchley, M.G., y S.A. Jarvis, *Concurrent Systems: Formal Development in CSP*, McGraw-Hill, 1995.

[JON91] Jones, C.B., *Systematic Development Using VDM*, 2.ª ed., Prentice-Hall, 1991.

[LIS86] Liskov, B.H., y V. Berzins, «An Appraisal of Program Specifications», publicado en *Software Specifications Techniques;* eds.: N. Gehani y A.T. McKittrick, Addison-Wesley, 1986, p. 3.

[MAR94] Marcianiak, J.J. (ed.), *Encyclopedia of Software Engineering*, Wiley, 1994.

[ROS95] Rosen, K.H., *Discrete Mathernatics and Its Applications*, 3.ª ed., McGraw-Hill, 1995.

[SPI88] Spievy, J.M., *Understanding Z: A Specification Language and Its Formal Semantics*, Cambridge University Press, 1988.

[SPI92] Spievy, J.M., *The Z Notation: A Reference Manual*, Prentice-Hall, 1992.

[WIL87] Witala, S.A., *Discrete Mathematics: A Unified Approach*, McGraw-Hill, 1987.

[WIN90] Wing, J.M., «A Specifier's Introduction to Formal Methods», *IEEE Computer*, vol. 23, n.º 9, Septiembre 1990, pp. 8-24.

[YOU94] Yourdon, E., «Formal Methods», *Guerrilla Programmer*, Cutter Information Corp., Octubre 1994.

PROBLEMAS Y PUNTOS A CONSIDERAR

25.1. Revisar los tipos de deficiencias asociados a los enfoques menos formales de la ingeniería del software en la Sección 25.1.1. Proporcione tres ejemplos de cada uno de ellos, procedentes de su propia experiencia.

25.2. Los beneficios de las matemáticas como mecanismo de especificación se han descrito con cierta extensión en este capítulo. ¿Existe algún aspecto negativo?

25.3. Se le ha asignado un equipo de software que va a desarrollar software para un fax módem. Su trabajo consiste en desarrollar el «listín telefónico» de la aplicación. La función del listín telefónico admite hasta *MaxNombre* nombres de direcciones que serán almacenados junto con los nombres de la compañía, números de fax y otras informaciones relacionadas. Empleando el lenguaje natural, defina:

a. el invariante de datos

b. el estado

c. las operaciones probables

25.4. Se le ha asignado un equipo de software que está desarrollando software, denominado *DuplicadosMemoria*, y que proporciona una mayor cantidad de memoria aparente para un PC, en comparación con la memoria física. Esto se logra identificando, recogiendo y reasignando bloques de memoria que hayan sido asignados a aplicaciones existentes pero no estén siendo utilizados. Los bloques no utilizados se reasignan a aplicaciones que requieran memoria adicional. Efectuando las suposiciones oportunas, y empleando el lenguaje natural, defina:

a. el invariante de datos

b. el estado

c. las operaciones probables

25.5. Desarrollar una especificación constructiva para un conjunto que contenga tuplas de números naturales de la forma (x, y, z^2) tales que la suma de x e y es igual a z.

25.6. El instalador de una aplicación basada en PC determina si está presente o no un conjunto de recursos de hardware y software adecuados. Comprueba la configuración de hardware para determinar **si** están presentes o no diferentes dispositivos (de entre muchos dispositivos posibles) y determina si ya están instaladas las versiones específicas de software del sistema y de controladores. ¿Qué conjunto de operadores se utilizaría para lograr esto? Proporcionar un ejemplo en este contexto.

25.7. Intente desarrollar una expresión empleando la lógica y un conjunto de operadores para la siguiente sentencia: «Para todo x e y, si x es padre de y e y es padre de z, entonces x es abuelo de z. Todas las personas tienen un padre.» Pista: utilice las funciones $P(x, y)$ y $G(x, z)$ para representar las funciones padre y abuelo, respectivamente.

25.8. Desarrollar una especificación constructiva de conjuntos correspondiente al conjunto de parejas en las cuales el primer elemento de cada pareja es la suma de dos números naturales no nulos, y el segundo elemento es la diferencia de esos dos números. Ambos números deben de estar entre 100 **y** 200 inclusive.

25.9. Desarrollar una descripción matemática del estado y del invariante de datos para el Problema 25.3. Refinar esta descripción en el lenguaje de especificación **Z.**

25.10. Desarrollar una descripción matemática del estado **y** del invariante de datos para el Problema 25.4. Refinar esta descripción en el lenguaje de especificación **Z.**

25.11. Utilizando la notación **Z** presentada en la Tabla 25.1, seleccionar alguna parte del sistema de seguridad *HogarSeguro* descrito anteriormente en este libro, e intentar especificarlo empleando **Z.**

25.12. Empleando una o más de las fuentes de información indicadas en las referencias de este capítulo o en la sección de *Otras Lecturas y Fuentes de Información,* desarrollar una presentación de media hora acerca de la sintaxis y semántica básica de un lenguaje de especificación formal y distinto de **Z.**

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Además de los libros utilizados como referencia en este capítulo, se ha publicado un número bastante grande de libros acerca de temas relacionados con **los** métodos formales a lo largo de los últimos años. Se presenta a continuación un listado de los ejemplos más significativos:

Bowan, **J.**, *Formal Specification and Documentation using Z: A Case study Approach,* International Thomson Computer Press, 1996.

Casey, **C.**, *A Programming Approach to Formal Methods,* McGraw-Hill, 2000.

Cooper, D., y R. Barden, *Z in Practice,* Prentice-Hall, 1995.

Craigen, D., S. Gerhart y T. Raltson, *Industrial Application of Formal Methods to Model, Design, Diagnose and Analize Computer Systems*, Noyes Data Corp., Park Ridge, NJ, 1995.

Diller, A., *Z.: An Introduction to Formal Methods,* 2.ª ed., Wiley, 1994.

Harry, A., *Formal Methods Fact File: VDM y Z*, Wiley, 1997

Hinchley, **M.,** y J. Bowan, *Applications of Formal Methods,* Prentice-Hall, 1995.

Hinchley, **M.,** y J. Bowan, *Industrial Strength Formal Methods,* Academic Press, 1997.

Hussmann, H., *Formal Foundations for Software Engineering Methods,* Springer-Verlag, 1997.

Jacky, J., The Way of Z: *Practica1 Programming With Formal Methods,* Cambridge University Press, 1997.

Lano, J., y Haughton (eds.), *Object-Oriented Specification Case Studies,* Prentice-Hall, 1993.

Rann, D., J. Turner y J. Whitworth, *Z: A Beginner's Guide,* Chapman & Hall, 1994.

Ratcliff, B., *Introducing Specification Using Z: A Practical Case Study Approach,* McGraw-Hill, 1994.

D. Sheppard, *An Introduction to Formal Specification with Z and VDM,* McGraw-Hill, 1995.

Los ejemplos de septiembre de 1990 de *IEEE Transactions on Software Engineering, IEEE Software e IEEE Computer* estaban dedicados todos ellos a los métodos formales. Siguen siendo una fuente excelente de información Útil.

Schuman ha editado un libro que describe los métodos formales y las tecnologías orientadas a objetos *(Formal Object-Oriented Development*, Springer-Verlag, 1996). **El** libro ofrece líneas generales acerca de la utilización selectiva de métodos formales y acerca de la forma en que estos métodos se pueden utilizar en conjunción con enfoques OO.

En Internet se encuentra una gran cantidad de información acerca de los métodos formales y de otros temas relacionados. En **http://www.pressman5.com** se puede encontrar una lista de referencias actualizada relevante para los métodos formales.

CAPÍTULO 26 INGENIERÍA DEL SOFTWARE DE SALA LIMPIA

LA utilización integrada del modelado de ingeniería del software convencional, métodos formales, verificación de programas (demostraciones de corrección) y estadística SQA se han combinado en una técnica que puede dar lugar a un software de calidad extremadamente alta. La *ingeniería del software de sala limpia* es un enfoque que hace hincapié en la necesidad de incluir la corrección en el software a medida que éste se desarrolla. En lugar del ciclo clásico de análisis, diseño, pruebas y depuración, el enfoque de sala limpia sugiere un punto de vista distinto [LIN94]:

> La filosofía que subyace tras la ingeniería del software de sala limpia consiste en evitar **la** dependencia de costosos procesos de eliminación de defectos, mediante la escritura de incrementos de código desde un primer momento, y mediante la verificación de su corrección antes de las pruebas. Su modelo de proceso incluye la certificación estadística de calidad de los incrementos de código, a medida que estos se van añadiendo cn el sistema.

En muchos aspectos, el enfoque de sala limpia eleva la ingeniería del software a otro nivel. Al igual que las técnicas de métodos formales que se presentaban en el Capítulo 25, el proceso de sala limpia hace hincapié en el rigor en la especificación y en el diseño, y en la verificación formal de cada uno de los elementos del modelo de diseño resultante mediante el uso de pruebas de corrección basadas en fundamentos matemáticos. Al extender el enfoque adoptado en los métodos formales, el enfoque de sala limpia hace hincapié también en técnicas de control estadístico de calidad, incluyendo las comprobaciones basadas en el uso anticipado del software por parte de los clientes.

VISTAZO RÁPIDO

¿Qué es? ¿Cuántas veces se ha oído decir «Hazlo correctamente a la primera.? Esa es la filosofía primordial de la ingeniería del software de sala limpia, un proceso que da importancia a la verificación matemática de la corrección antes de que comience la construcción de un programa y de que la certificación de la fiabilidad del software forme parte de la actividad de pruebas. Haciendo hincapié en una filosofía más profunda, se trataría de aquella que tiene índices de fallo extremadamente bajos y que es dificil o imposible de lograr utilizando métodos menos formales.

¿Quién lo hace? Un ingeniero del software formado para esta especialización.

¿Por qué es importante? Los errores conllevan doble trabajo. Trabajar el doble lleva más tiempo y es más caro. ¿No sería maravilloso poder reducir dramáticamente la cantidad de errores (fallos informáticos) que se cometen en el diseño y construcción del software? Esto es lo que promete la ingeniería del software **de** sala limpia.

¿Cuáles son los pasos? Los modelos de análisis y diseño se crean utilizando la representación de estructura de caja. Una «caja» encapsula el sistema (o algún aspecto del sistema) a un nivel específico de abstracción. La verificación de la corrección se aplica una vez que se ha completado el diseño de la estructura de caja Y la prueba estadística de la utilización comienza una vez que se ha verificado la corrección en cada estructura de caja. El software se prueba definiendo un conjunto de escenarios, determinando la probabilidad de utilización de cada uno y definiendo entonces las pruebas aleatorias que se ajustan a las probabilidades. Por último, los registros de errores resultantes se analizan para permitir el cálculo matemático de la fiabilidad proyectada en el componente de software.

¿Cuál es el producto obtenido? El desarrollo de especificaciones de caja negra, de caja de estado y de caja limpia. Y, además, el registro de los resultados de las pruebas formales de corrección y las pruebas estadísticas de utilización.

¿Cómo puedo estar seguro de que lo he hecho correctamente? La prueba formal de corrección se aplica a la especificación de estructura de cajas. Las pruebas estadísticas de utilización ejercitan los escenarios de utilización para asegurar que no se revelen y se puedan corregir los errores en la funcionalidad del usuario. Los datos de prueba se utilizan para proporcionar una señal de la fiabilidad del software.

Cuando el software falla en el mundo real, suelen abundar los peligros a largo plazo así como los peligros inmediatos. Los peligros pueden estar relacionados con la seguridad humana, con pérdidas económicas o con el funcionamiento efectivo de una infraestructura social y de negocios. La ingeniería del software de sala limpia es un modelo de proceso que elimina los defectos antes de que puedan dar lugar a riesgos graves.

26.1 EL ENFOQUE DE SALA LIMPIA

La filosofía de la «sala limpian en las técnicas de fabricación de hardware es en realidad algo bastante sencillo: se trata de una forma rentable y eficiente, en términos de tiempo, de establecer un enfoque de fabricación que impida la introducción de defectos de producción. En lugar de fabricar un producto y dedicarse después a eliminar defectos, el enfoque de sala limpia demanda la disciplina necesaria para eliminar errores en las especificaciones y en el diseño, fabricando entonces el producto de forma «limpia».

Cita:

La ingeniería de sala limpia logra control de calidad estadístico sobre el desarrollo del software separando estrictamente el proceso del diseño del proceso de comprobación en un cauce de desarrollo incremental de software.

Harlan Mills

La filosofía de sala limpia fue propuesta por primera vez para la ingeniería del software por parte de Mills y sus colegas [WIL87] durante los años 80. Aun cuando las primeras experiencias acerca de este enfoque disciplinado para los trabajos relacionados con el software mostraba promesas significativas [HAU94], no ha alcanzado una amplia utilización. Henderson [HEN95] sugiere tres posibles razones:

1. La creencia en que la metodología de sala limpia es excesivamente teórica, excesivamente matemática y excesivamente radical para utilizarla en el desarrollo de software real.
2. No propugna una comprobación unitaria por parte de los desarrolladores, sino que la sustituye por una verificación de la corrección y por un control estadístico de la calidad —estos conceptos que representan una desviación fundamental con respecto a la forma en que se desarrolla la mayor parte del software en la actualidad—.
3. La madurez de la industria de desarrollo del software. El uso de procesos de sala limpia requiere una aplicación rigurosa de procesos definidos en todas las fases del ciclo de vida. Dado que la mayor parte de la industria funciona todavía en el nivel ad hoc (según se ha definido por parte del Software Engineering Institute Capability Maturity Model), la industria no ha estado preparada para aplicar estas técnicas.

Aun cuando existen ciertos indicios de verdad en todas las indicaciones expresadas anteriormente, los posibles beneficios de la ingeniería del software de sala limpia compensan más que sobradamente la inversión requerida para superar la resistencia cultural que se encuentra en el núcleo de estas objeciones.

26.1.1. La estrategia de sala limpia

El enfoque de sala limpia hace uso de una versión especializada del modelo incremental de software (Capítulo 2). Se desarrolla un «cauce de incrementos de software» [LIN94] por parte de equipos de ingeniería del software pequeños e independientes. A medida que se va certificando cada incremento, se integra en el todo. Consiguientemente, la funcionalidad del sistema va creciendo con el tiempo.

¿Cuáles son las tareas principales que se realizan en la ingeniería del software de sala limpia?

La sucesión de tareas de sala limpia para cada incremento se ilustra en la Figura 26.1. Los requisitos globales del sistema o producto se van desarrollando empleando los métodos de ingeniería de sistemas descritos en el Capítulo 10. Una vez que se ha asignado una funcionalidad al elemento de software del sistema, el tubo de la sala limpia comienza sus incrementos. Se producen las tareas siguientes:

Planificación de incrementos. Se desarrolla un plan de proyecto que adopta la estrategia incremental. Se van estableciendo las funcionalidades de cada uno de los incrementos, su tamaño estimado y un plan de desarrollo de sala limpia. Es preciso tener especial cuidado para asegurar que los incrementos certificados se vayan integrando de forma temporalmente oportuna.

Recolección de requisitos. Mediante el uso de técnicas similares a las presentadas en el Capítulo 11, se desarrolla una descripción más detallada de requisitos del nivel del usuario (para cada incremento).

Especificación de la estructura de cajas. Se utiliza un método de especificación que hace uso de estructuras de caja [HEV93] para describir la especificación funcional.

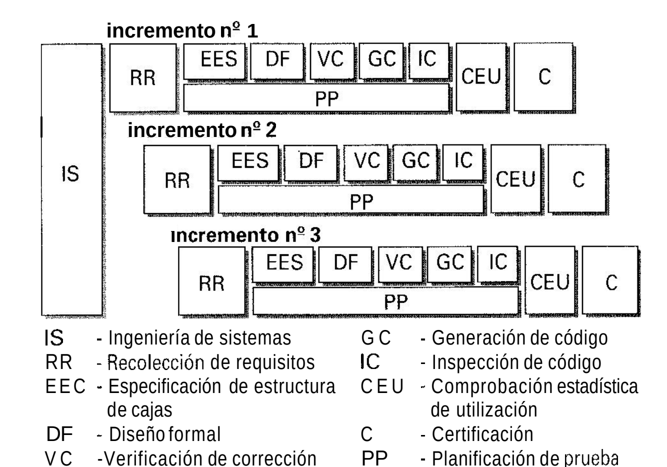


FIGURA 26.1. El modelo de proceso de sala limpia.

Ajustado a los principios de análisis operacional descritos en el Capítulo 11, la estructura de caja «aísla y separa la definición creativa del comportamiento, de los datos, y de los procedimientos para cada nivel de refinamiento».



Uno fuente excelente de información y de recursos poro la ingeniería del software de sala limpio se puede entontror en **www.tleansoft.com**

Diseño formal. Mediante el uso del enfoque de estructura de cajas, el diseño de sala limpia es una extensión natural y sin discontinuidades de la especificación. Aun cuando es posible efectuar una distinción clara entre estas dos actividades, las especificaciones (que se denominan «cajas negras») se refinan iterativamente (dentro de cada incremento) para transformarse en diseños análogos a la arquitectura y a los procedimientos (que se denominan «cajas de estado» y «cajas trasparentes», respectivamente).

Verificación de corrección. El equipo de sala limpia lleva a cabo una serie de rigurosas actividades de verificación de corrección aplicadas primero al diseño y después al código. La verificación (Secciones 26.3 y 26.4) comienza con la estructura de cajas del más alto nivel (la especificación) y avanza hacia el detalle de diseño y el código. El primer nivel de verificación de corrección se lleva a cabo aplicando un conjunto de cuestiones de corrección» [LIN88]. Si este conjunto de preguntas no demuestra que la especificación es correcta, se utilizan métodos más formales (matemáticos) de verificación.



Generación de código, inspección y verificación. Las especificaciones de estructura de caja, que se representan mediante un lenguaje especializado, se traducen al lenguaje de programación adecuado. Se utilizan entonces técnicas estándar de recorrido o de inspección (Capítulo 8) para asegurar el cumplimiento semántico de las estructuras de código y de cajas, y la corrección sintáctica de código. A continuación, se efectúa una verificación de corrección para el código fuente.

Planificación de la comprobación estadística. La utilización estimada del software se analiza, se planifica y se diseña un conjunto de casos de prueba que ejerciten la «distribución de probabilidad» de esa utilización (Sección 26.4). Según se muestra en la Figura 26.1, esta actividad de sala limpia se realiza en paralelo con la especificación, la verificación y la generación de código.



la sala limpia da importancia a las pruebas que ejercitan la manera en que se utilizo realmente el software. los casos de utilización proporcionan una fuente excelente para el proceso de planificación estadística de comprobación.

Comprobación estadística de utilización. Recordando que es imposible una comprobación exhaustiva del software de computadora (Capítulo 17), siempre resulta necesario diseñar un conjunto finito de casos de prueba. Las técnicas estadísticas de utilización [POO88] ejecutan una colección de pruebas derivadas de una muestra estadística (la distribución de probabilidad indicada anteriormente) de todas las posibles ejecuciones del programa por parte de todos los usuarios de una cierta población objetivo (Sección 26.4).

Certificación. Una vez que se ha finalizado la verificación, la inspección y la comprobación de utilización (y después de corregir todos los errores) se certifica el incremento como preparado para su integración.

Al igual que otros modelos de proceso del software descritos en otras partes de este libro, el proceso de sala limpia hace especial hincapié en la necesidad de conducir unos modelos de análisis y de diseño de muy alta calidad. Según se verá posteriormente en este capítulo, la notación de estructura de cajas no es más que otra forma para que el ingeniero del software pueda representar los requisitos y el diseño. La distinción real del enfoque de sala limpia consiste en que se aplica la verificación formal a los modelos de ingeniería.

26.1.2. ¿Qué hace diferente la sala limpia?

Dyer [DYE92] alude a las diferencias del enfoque de sala limpia cuando define el proceso:

> La sala limpia representa el primer intento práctico de poner el proceso de desarrollo del software bajo un control estadístico de calidad con una estrategia bien definida para la mejora continua del proceso. Para alcanzar esta meta, sc definió un ciclo único de vida de sala limpia, que hacía hincapié en una ingeniería del software basada en las matemáticas para obtener diseños de software correctos y que se basaba en software basado en estadística para ***la*** certificación de fiabilidad de ese software.

La ingeniería del software de sala limpia ditiere de los puntos de vista convencionales y orientados a objetos que se representan en la Partes Tercera y Cuarta de este libro porque:

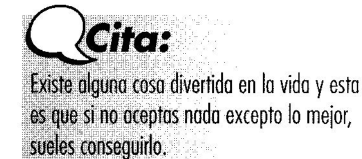
1. Hace uso explícito del control estadístico de calidad.
2. Verifica la especificación del diseño empleando una demostración de corrección basada en las matemáticas.
3. Hace mucho uso de la comprobación estadística de utilización para descubrir errores de especial incidencia.



Las características más importantes que distinguen la sala limpia son la comprobación de corrección y la comprobación estadística de utilización.

Evidentemente, el enfoque de sala limpia aplica la mayor parte, si es que no todos, de los principios básicos de ingeniería del software y de los conceptos que se han presentado a lo largo de este libro. Son esenciales unos buenos procedimientos de análisis y diseño si es que se desea producir una elevada calidad. Pero la ingeniería de sala limpia diverge de las prácticas de software convencionales al quitar importancia (hay quien diría eliminar) al papel de las pruebas de unidad y a la depuración y al reducir dramáticamente (o eliminar) la cantidad de comprobaciones que son realizadas por quien desarrolla el software[1].

En el desarrollo convencional del software, los errores se aceptan como cosas que pasan. Dado que se considera que los errores son inevitables, cada módulo del programa debe ser comprobado unitariamente (para descubrir los errores) y depurado después (para eliminar los errores). Cuando se publica finalmente el software, la utilización práctica descubre aun más defectos, y comienza otro ciclo de comprobación y depuración. Este trabajo repetido asociado a las actividades mencionadas resulta costoso y lleva mucho tiempo. Y lo que es peor, puede ser degenerativo; la corrección de errores puede (inadvertidamente) dar lugar a la introducción de otros errores.



En la ingeniería del software de sala limpia, la comprobación unitaria y la depuración se ven sustituidas por una verificación de corrección y por pruebas basadas en la estadística. Estas actividades, junto con el mantenimiento de registros para una continua mejora, hacen que el enfoque de sala limpia sea único.

26.2 ESPECIFICACIÓN FUNCIONAL

Independientemente del método de análisis seleccionado, los principios de operación presentados en el Capítulo 11 siempre serán aplicables. Se modelan los datos, las funciones y el comportamiento. Los modelos resultantes deben de ser descompuestos (refinados) para proporcionar un grado de detalle cada vez más elevado. El objetivo global consiste en pasar de una especificación que captura la esencia de un problema, a una especificación que proporciona una cantidad de detalle sustancial para su implementación.

La ingeniería del software de sala limpia satisface los principios de análisis operacional por cuanto emplea un método denominado *especificación de estructuru de caja*. Una «caja» encapsula el sistema (o algún aspecto del sistema) con un cierto grado de detalle. Mediante un proceso de refinamiento progresivo, se van refinando las cajas para formar una jerarquía en la cual cada caja tiene *transparencia referencial.* Esto es, «el contenido de información de cada especificación de caja basta para definir su refinamiento, sin depender de la implementación de ninguna otra caja» [LIN94]. Esto capacita al analista para desglosar jerárquicamente el sistema, pasando de la representación esencial situada en la parte superior, hasta los detalles específicos de la implementación situados en la parte inferior. Se utilizan tres tipos de cajas:

Caja negra. Esta caja especifica el comportamiento del sistema, o de parte de un sistema. El sistema (o parte de él) responde a estímulos específicos (sucesos) mediante la aplicación de un conjunto de reglas de transición que hacen corresponder el estímulo con la respuesta.

Caja de estado. Esta caja encapsula los datos de estados y de servicios (operaciones) de forma análoga a los objetos. En esta vista de especificación, se representan las entradas de la caja de estados (los estímulos) y sus salidas (las respuestas). La caja de estados también representa la «historia de estímulos» de la caja negra, es decir, los datos encapsulados en la caja de estado que deben ser mantenidos entre las transiciones implicadas

Caja limpia. Las funciones de transición que están implicadas en la caja de estado se definen en la caja limpia. Dicho literalmente, la caja limpia contiene el diseño procedimental correspondiente a la caja de estados.

? ¿Cómo se lleva o cabo el refinamiento como parte de lo especificación de estructura de caja negra?

[1] Las pruebas se realizan, pero las efectúa un equipo independiente de pruebas

La Figura 26.2 ilustra el enfoque de refinamiento mediante el uso de una especificación de estructura de cajas. Una caja negra (CN,) define las respuestas de todo un conjunto completo de estímulos. CN, se puede refinar en un conjunto de cajas negras, desde CN, , hasta CN, ,,,cada una de las cuales aborda una cierta clase de comportamiento. El refinamiento prosigue hasta que se identifique una clase cohesiva de comportamiento (por ejemplo, CN, ,,). A continuación, se define una caja de estado (CE, ,,) para la caja negra (CN, ,,). En este caso, CE, , , contiene todos los datos y servicios necesarios para implementar el comportamiento definido por CN, ! ,. Por último, se refina CE, , , para formar un conjunto de cajas transparentes (CT, , , n) y se especifican los detalles de diseño de procedimientos.

A medida que se va realizando cada uno de estos pasos de refinamiento, se produce también una verificación de la corrección. Se verifican las especificaciones de las cajas de estado para asegurar que todas y cada una de ellas se ajustan al comportamiento definido por la especificación de la caja negra predecesora. De manera similar, se verifican las especificaciones de las cajas transparentes con respecto a la caja de estados predecesora.



CLAVE

El refinamiento de la estructura de cajas y la verificación de corrección ocurren simultáneamente.

Es preciso tener en cuenta que los métodos de especificación basados en métodos formales (Capítulo 25) se pueden utilizar en lugar del enfoque de especificación de estructura de cajas. El único requisito *es* que se puede verificar formalmente cada uno de los niveles de especificación.

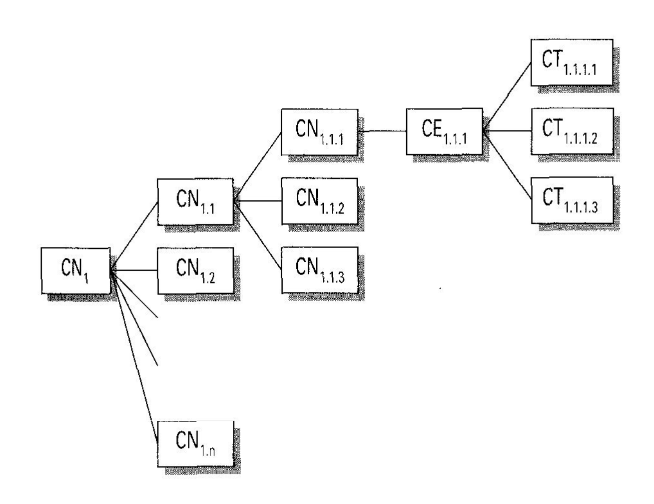


FIGURA 26.2. Refinamiento de la estructura de cajas.

26.2.1. Especificación de caja negra

Una especificación de caja negra describe una abstracción, estímulos y respuestas empleando la notación que se muestra en la Figura 26.3. [MIL88]. La función *f* se aplica a una secuencia, *S**, de entradas (estímulos) **y** esta función los transforma en una salida (respuesta), *R*. Para componentes sencillos de software, *f*, puede ser una función matemática, pero en general, *f* se describe empleando el lenguaje natural *(o* bien un lenguaje de especificación formal).

Muchos de los conceptos introducidos para sistemas orientados a objetos son aplicables también para la caja negra. Las abstracciones de datos y las operaciones que manipulan estas abstracciones, se ven encapsuladas por la caja negra. Al igual que una jerarquía de clases, la especificación de caja negra puede mostrar las jerarquías de utilización en que las cajas de nivel inferior heredan las propiedades de las cajas de nivel superior dentro de la estructura de árbol.

Referencia cruzada

los conceptos orientados a objetos se describen en el Capítulo 20.

26.2.2. Especificación de caja de estado

La caja de estado es «una generalización sencilla de una máquina de estado» [MIL88]. Recordando la descripción del modelado de comportamiento y de los diagramas de transición de estados que se ofrece en el Capítulo 12, un estado es algún modo observable de comportamiento del sistema. A medida que se produce el procesamiento, el sistema va respondiendo a sucesos (estímulos) efectuando una transición que parte del estado y llega a algún nuevo estado. A medida que se efectúa la transición, puede producirse una acción. La caja de estado utiliza una abstracción de datos para determinar la transición al estado siguiente (respuesta) que se producirá como consecuencia de la transición.

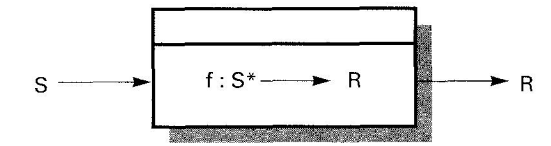


FIGURA 26.3. Una especificación de caja negra [MIL88].

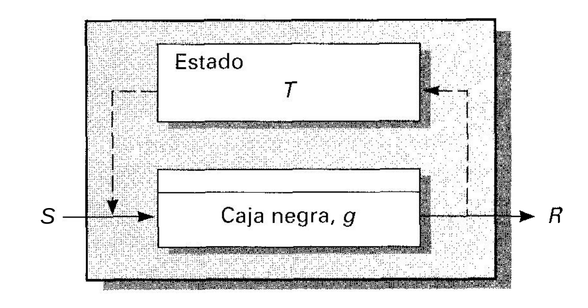


FIGURA 26.4. Una especificación de caja de estado [MIL88].

Según se muestra en la Figura 26.4, la caja de estado contiene una caja negra. El estímulo, S, que se introduce en la caja negra, procede de alguna fuente externa y de un conjunto de estados internos del sistema, T. Mills proporciona una descripción matemática de la función, f, de la caja negra contenida en el seno de la caja de estado:

$$g: S^* \times T^* \rightarrow R \times T$$

donde g es una subfunción que está asociada a un estado específico t. Cuando se consideran en su conjunto, las parejas de estados y subfunciones (t, g) definen la función de caja negra f.

26.2.3. Especificación de caja limpia

La especificación de caja limpia está íntimamente relacionada con el diseño de procedimientos y con la programación estructurada. En esencia, la subfunción g, que se encuentra dentro de la caja de estado, se ve sustituida por las estructuras de programación estructurada que implementa g.

Referencia cruzada

El diseño de procedimientos y la programación estructurada se describen en el Capítulo 16.

Como ejemplo, considérese la caja limpia que se muestra en la Figura 26.5. La caja negra g, que se muestra en Figura 26.4 se ve sustituida por una sucesión de estructuras que contienen una estructura condicional. Éstas, a su vez, se pueden refinar para formar cajas transparentes del interior a medida que vaya avanzando el procedimiento de refinamiento progresivo.

Es importante tener en cuenta que la especificación de procedimientos descrita en la jerarquía de caja limpia se puede demostrar a efectos de corrección. Este tema se considerará en la sección siguiente.

26.3 REFINAMIENTO Y VERIFICACIÓN DEL DISEÑO

El enfoque de diseño que se utiliza en la ingeniería del software de sala limpia hace mucho uso de la filosofía de la programación estructurada. Pero, en este caso, la programación estructurada se aplica de forma mucho más rigurosa.

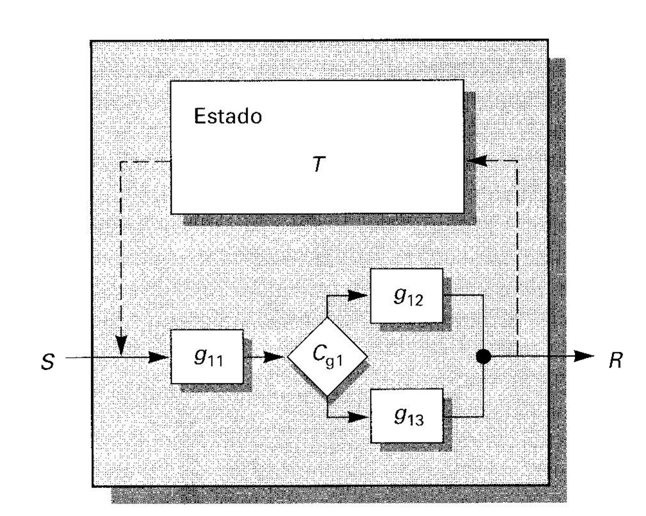


FIGURA 26.5. Una especificación de caja transparente.

La funciones básicas de procesamiento (que se describían durante los refinamientos anteriores de la especificación) se refinan ahora utilizando una «expansión progresiva de funciones matemáticas en estructuras de conectividad lógica [por ejemplo, *si-entonces-sino]* y subfunciones, donde la expansión [se] efectúa hasta que todas las funciones identificadas pudieran ser enunciadas directamente en el lenguaje de programación utilizado para la implementación» [DYE92].

El enfoque de la programación estructurada se puede utilizar de forma eficiente para refinar la función, pero ¿qué pasa con el diseño de datos'? En este aspecto, entra en juego un cierto número de conceptos fundamentales de diseño (Capítulo 13). Los datos del programa se encapsulan como un conjunto de abstracciones a las cuales prestan servicio las subfuncionec. Los conceptos de encapsulamiento de datos, ocultamiento de información y los tipos de datos se utilizan entonces para crear el diseño de datos.



El programa DoD STARTS ha desarrollado varias guías y documentos de sala limpio: **ftp.cdrom.com/pub/ada/docs/cleanrm**

26.3.1. Refinamiento y verificación del diseño

Cada especificación de caja limpia representa el diseño de un procedimiento (subfunción) necesario para efectuar una transición de caja de estado. Mediante la caja limpia, se utilizan las estructuras de programación estructurada y de refinamiento progresivo según se ilustra en la Figura 26.6. Una función de programa, f, se refina para dar lugar a una sucesión de subfunciones g y h. Éstas a su vez se refinan para formar estructuras condicionales *(si-entonces-sino* y *hacer-mientras).* Un refinamiento posterior ilustra la elaboración lógica continuada.

¿ Qué condiciones se aplican para probar que las construcciones estructuradas son correctas?

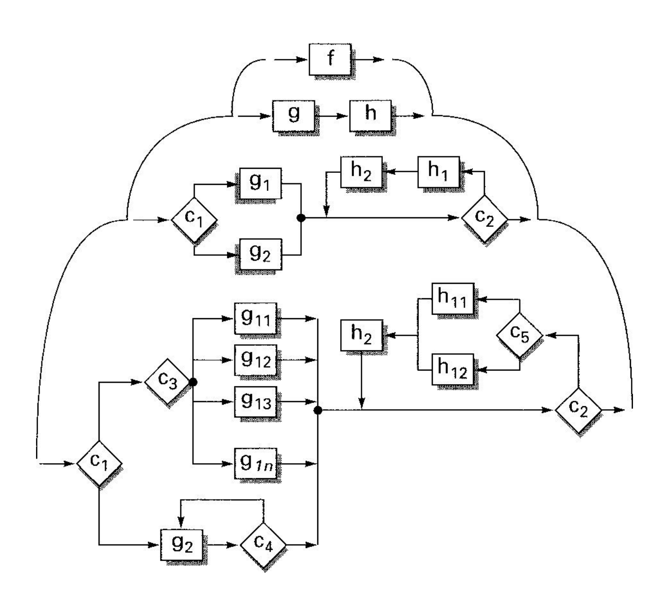


FIGURA 26.6. Refinamiento progresivo.

En cada nivel de refinamiento, el equipo de sala limpia[2] lleva a cabo una verificación formal de corrección. Para lograr esto, se asocia un conjunto de *condiciones de corrección* genéricas a las estructuras de programación estructurada. Si se expande una función *f* para dar una sucesión *g* y *h*, entonces la condición de corrección para todas las entradas de *f* es:

- **¿Es cierto que *g* seguido por *h* da lugar a *f*?**

  Si se refina una función *p* para llegar a una estructura condicional *(si-entonces-sino)*, la condición de corrección para toda entrada de *p* es:

- **¿Siempre que sea cierta la condición (c) es cierto que *q* realiza p y siempre que (c) sea falsa, es cierto que Y realiza p?**

  Si se refina una función m en forma de bucle, las condiciones de corrección para todas las entradas de *m* son:

- **¿Está garantizada la finalización?**
- **¿Siempre que (c) sea verdadera es cierto que *n* seguido por *m* realiza *m*, siempre que (c) sea falso, sigue realizándose *m* si se obvia el bucle?**



Si nos limitamos o construcciones estructuradas cuando se creo un diseño de procedimientos, lo pruebo de lo corrección es sencillo. Si se violan los construcciones, los pruebas de corrección son difíciles o imposibles.

Cada vez que se refina una caja limpia hasta el siguiente nivel de detalle, se aplican las condiciones de corrección indicadas anteriormente.

Es importante tener en cuenta que la utilización de estructuras de programación estructurada restringe el número de comprobaciones de corrección que es preciso efectuar. Se verifica una sola condición en busca de sucesiones; se verifican dos condiciones para los si-entonces-sino; y se verifican tres condiciones para los bucles.

Con objeto de ilustrar alguna verificación de corrección para un diseño de procedimientos, se utiliza un sencillo ejemplo presentado por primera vez por parte de Linger y sus colaboradores [LYN79]. Nuestro objetivo es diseñar y verificar un pequeño programa que calcule la parte entera, y, de una raíz cuadrada de un entero dado, *x*. El diseño de procedimientos se representa empleando el diagrama de flujo de la Figura 26.7.

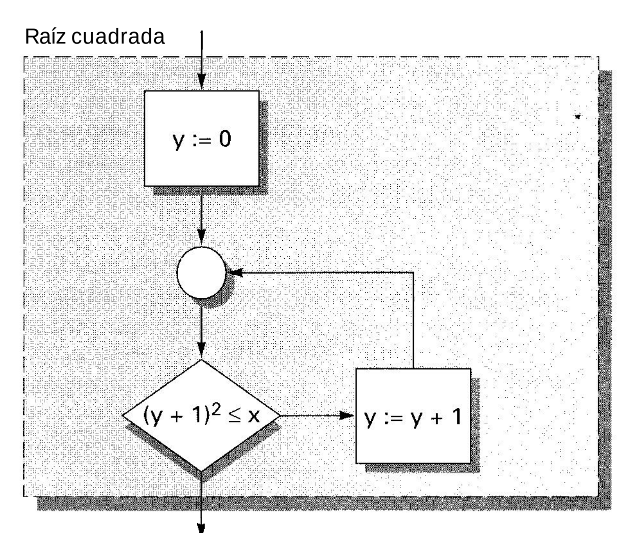


FIGURA 26.7. Cálculo de la parte entera de una raíz cuadrada [LIN79].

Para verificar la corrección de este diseño, es preciso definir las condiciones de entrada y de salida que se indican en la Figura 26.8. La condición de entrada indica que *x* debe ser mayor o igual a 0. La condición de salida requiere que *x* permanezca intacta y que adopte un valor dentro del intervalo mostrado en la figura. Para demostrar que el diseño es correcto, es necesario demostrar que las condiciones *comienzo*, *bucle*, *cont*, *si* y *salida* mostradas en la Figura 26. 8 son ciertas en todos los casos. En algunas ocasiones se denominan *subdemostraciones*.

[2] Dado que todo el equipo está implicado en el proceso de verificación, es menos probable que se produzca un error al efectuar la verificación en *sí*.

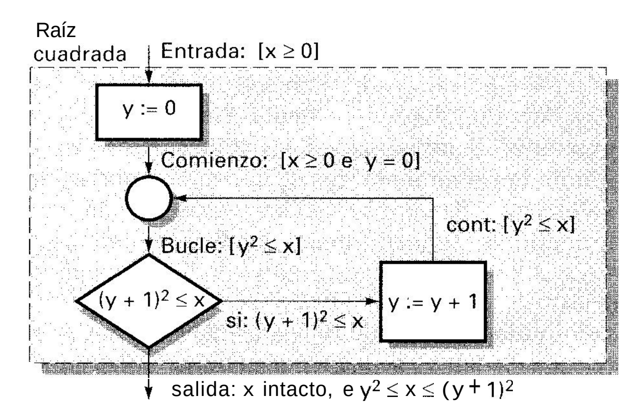


FIGURA 26.8. Demostración de la corrección del diseño [LIN79].

1. La condición *comienzo* exige que [$x \geq 0$ e $y = 0$]. Basándose en los requisitos del problema, se supone que la condición de entrada es correcta[3]. Consiguientemente, se satisface la primera parte de la condición *comienzo*, $x \geq 0$. En el diagrama de flujo, la sentencia que precede inmediatamente a la condición *comienzo* hace que $y = 0$. Consiguientemente, la segunda parte de la condición *comienzo* también se satisface. De aquí que *comienzo* sea verdadero.
2. La condición *bucle* se puede encontrar de dos maneras: (1) directamente saliendo de *comienzo* (en este caso, la condición del *bucle* se satisface directamente) o bien (2) a través del control de flujo que pasa por la condición *cont*. Dado que la condición *cont* es idéntica a la condición *bucle*, *bucle* es verdadera independientemente de la rama de flujo que lleve a ella.



Para demostrar la corrección de un diseño, primero se deben identificar todas las condiciones y demostrar que toda una de ellas toma un valor Booleano. Esto es lo que se llama subdemostración.

3. Se llega a la condición *cont* únicamente después de haber incrementado en 1 el valor de *y*. Además, la ruta de flujo de control que lleva a *cont* solamente se puede invocar si la condición *si* también es verdadera. Consiguientemente si $(y+1)^2 \leq x$, se sigue que $y^2 \leq x$. La condición *cont* se satisface.
4. La condición *si* se comprueba en la lógica condicional que se muestra. Consiguientemente, la condición *si* debe de ser verdadera cuando el flujo de control pase por la vía mostrada.
5. La condición *salido* exige, en primer lugar, que *x* no haya cambiado. Un examen del diseño indica que *x* no aparece en ningún lugar a la izquierda de un operador de asignación. No hay ninguna llamada a función que utilice x. Por tanto, queda intacto. Dado que la comprobación de condiciones $(y+1)^2 \leq x$ tiene que fallar para alcanzar la condición *solido*, se sigue que $(y+1)^2 \leq x$. Además, la condición *bucle* debe seguir siendo verdadera (esto es, $y^2 \leq x$). Consiguientemente, que $(y+1)^2 > x$ e $y^2 \leq x$ se pueden combinar para satisfacer la condición de *salida*.

Además, es preciso asegurar que finalice el bucle. Un examen de la condición del bucle indica que dado que *y* se va incrementando y que $x \geq 0$, el bucle finalmente debe terminar.

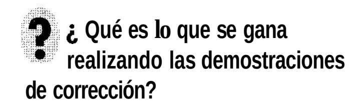
Los cinco pasos indicados anteriormente son una demostración de la corrección del diseño del algoritmo indicado en la Figura 26.7. Ahora estamos seguros de que el diseño calculará, realmente, la parte entera de una raíz cuadrada.

Es posible utilizar un enfoque matemáticamente más riguroso de la Verificación del diseño. Sin embargo, un debate sobre este tema iría más allá del alcance de este libro. Los lectores interesados pueden consultar [LIN79].

26.3.2. Ventajas de la verificación del diseño[4]

La verificación de corrección rigurosa de cada uno de los refinamientos del diseño de caja limpia posee un cierto número de ventajas evidentes. Linger [LIN94] las describe de la siguiente manera:

- *Se reduce la verificación a un proceso finito.* La forma anidada y secuencial, en que se organizan las estructuras de control en una caja limpia, define de manera natural una .jerarquía que revela las condiciones de corrección que es preciso verificar. Un axioma de sustitución [LIN79] nos permite reemplazar las funciones objetivo por sus refinamientos de estructura de control dentro de la jerarquía de subdemostraciones. Por ejemplo, la subdemostracióii para la función final f1 de la Figura 26.9 requiere que la comprobación de las operaciones g1 y g2 con la función final f2 produzca sobre los datos el mismo efecto f1. Obsérvese que f2 reemplaza a todos los detalles de su refinamiento dentro de la demostración. Esta sustitución localiza el argumento de demostración en la estructura de control que se está estudiando. De hecho, permite al ingeniero del software realızar demostraciones en cualquier orden.



¿ Qué es lo que se gana realizando las demostraciones de corrección?

- *Es imposible asociar una importancia excesiva al efecto positivo que posee sobre la calidad lo reducción de la verificación a un proceso finito.* Aun cuando todos, salvo los programas más triviales, muestran un conjunto, que en esencia es infinito, de

[3] Un valor negativo para una raia cuadrada carece de significado en este contexto.

[4] Esta seccion y las Figuras 26 7 hasta la 26 9 se han adaptado de [LIN94] Se utilizan con permiso del autor

rutas de ejecución, se pueden verificar en un número finito de pasos.



Aun cuando en un proyromo hay uno contidod extremodomente grande de formas de ejecución, los posos poro demostrar que un proyromo es correcto son muy pocos.

```
[f1]                 f1 = [DO g1; g2; [f2] ENDI ?
  DO
   g1
   g2
   [f2]              f2 = [WHILE p1 DO [f3] ENDI ?
     WHILE
       p1
    DO [f3]          f3 = [DO g3; [f4]; g8 ENDI ?
       g3
       [f4]          f4 = [IF p2; THEN [f5] ELSE [f6] ENDI ?
        IF
          p2
       THEN [f5]     f5 = [DO g4; g5 ENDI ?
          g4
          g5
       ELSE [f6]     f6 = [DO g6; g7 ENDI ?
          g6
          g7
        END
       g8
    END
END
```

FIGURA 26.9. Diseño con subdemostraciones [LIN94].

- *Permite que los equipos de sala limpia verifiquen todas las líneas de diseño y de código.* Los diseños pueden efectuar la verificación mediante un análisis y debate en grupo de las bases del teorema de corrección y pueden producir pruebas escritas cuando se necesite una confianza adicional en algún sistema crítico para vidas o misiones.
- *Da lugar a un nivel de defectos próximo a cero.* Durante una revisión en equipo, se verifica por turnos la corrección de todas y cada una de las estructuras de control. Cada miembro del equipo debe estar de acuerdo en que cada condición es correcta, por tanto, un error solamente es posible si todos y cada uno de los miembros del equipo verifican incorrectamente una condición. El requisito de acuerdo unánime basada en las verificaciones individuales de resultados da lugar a un software que posee pocos o ningún defecto antes de su primera ejecución.
- *Es escalable.* Todo sistema de software, independientemente de su tamaño, posee unos procedimientos de caja transparente del más alto nivel formados por estructuras de secuencia, alternancias e iteraciones. Cada uno de estos invoca típicamente a un gran subsistema que posee miles de líneas de código —cada uno de estos subsistemas posee su propio nivel superior de funciones y procedimientos finales—. Por tanto, las condiciones de corrección para estas estructuras de control de alto nivel se verifican de la misma forma en que se procede con las estructuras de bajo nivel. Las verificaciones de alto nivel pueden requerir, y merecerá la pena, una mayor cantidad de tiempo, pero no se necesita más teoría.
- *Produce un código mejor que la comprobación unitaria.* La comprobación unitaria solamente comprueba los efectos de ejecutar vías de pruebas seleccionadas entre muchas vías posibles. Al basar la verificación en la teoría de funciones, el enfoque de sala limpia puede verificar todos y cada uno de los posibles efectos sobre los datos, porque aun cuando un programa pueda tener múltiples vías de ejecución, solamente posee una función. La verificación es, además, más eficiente que la comprobación unitaria. La mayor de las condiciones de verificación se pueden verificar en unos pocos minutos, pero las comprobaciones unitarias requieren una cantidad notable de tiempo para prepararlas, ejecutarlas y comprobarlas.

Es importante tener en cuenta que la verificación de diseño debe de aplicarse en última instancia al código fuente en sí. En este contexto, suele denominarse *verficación de corrección.*

26.4 PRUEBA DE SALA LIMPIA

La táctica y estrategia de la prueba de sala limpia es algo fundamentalmente distinto de los enfoques convencionales de comprobación. Los métodos convencionales derivan de casos de prueba para descubrir errores de diseño y de codificación. El objetivo de los casos de prueba de sala limpia es validar los requisitos del software mediante la demostración de que una muestra estadística de casos prácticos (Capítulo 11) se han ejecutado con éxito.

26.4.1. Prueba estadística de casos prácticos

El usuario de un programa de computadora no suele necesitar comprender los detalles técnicos del diseño. El comportamiento visible para el usuario de ese programa está controlado por las entradas y sucesos que suelen ser producidos por el usuario. Pero en casos complejos, el espectro posible de entradas y sucesos (esto es, los casos prácticos) pueden ser extremadamente variables. ¿Cuál es el subconjunto de casos prácticos que verifica adecuadamente el comportamiento del programa? Ésta es la primera cuestión que aborda la prueba estadística de casos prácticos.

La prueba estadística de casos «equivale a probar el software en la forma en que los usuarios tienen intención de utilizarlo» [LIN94]. Para lograr esto, los *equipos de prueba de sala limpia* (también llamados *equipos de certificación*) deben determinar la distri-

bución de probabilidad de utilización correspondiente al software. La especificación (caja negra) de cada incremento del software se analiza para definir un conjunto de estímulos (entradas o sucesos) que pueden dar lugar a que el software modifique su comportamiento. Basándose en entrevistas con posibles usuarios, en la creación de escenarios de utilización y en una comprensión general del dominio de la aplicación, se asigna una probabilidad de utilización a cada uno de los estímulos.

Los casos prácticos se generan para cada uno de los estímulos[5] de acuerdo con la distribución de probabilidad de utilización. Como ejemplo, considérese el sistema de seguridad *HogarSeguro* descrito anteriormente en este libro. Se está utilizando la ingeniería del software de sala limpia para desarrollar un incremento del software que gestione la interacción del usuario con el teclado del sistema de seguridad. Para este incremento se pueden identificar cinco estímulos. El análisis indica el porcentaje de probabilidad de cada estímulo. Para hacer que sea más sencilla la selección de casos de prueba, estas probabilidades se hacen corresponder con intervalos numerados entre 1 y 99 [LIN94], lo que se muestra en la tabla siguiente:

Estímulos del programa	Probabilidad	Intervalo
habilitar/ deshabilitar (HD)	50%	1-49
fijar zona (FZ)	15%	50-63
consulta (C)	15%	64-78
prueba (P)	15%	79-94
alarma (A)	5%	95-99

Para generar una sucesión de casos de prueba de utilización que se ajuste a la distribución de probabilidades de utilización, se genera una serie de números aleatorios entre 1 y 99. El número aleatorio corresponde al intervalo de distribución de probabilidad anteriormente destacado. Consiguientemente, la sucesión de casos prácticos se define aleatoriamente pero se corresponde con la probabilidad correspondiente de aparición de ese estímulo. Por ejemplo, suponga que se generan las siguientes sucesiones de números aleatorios

13–94–22–24–45–56
81–19–31–69–45–9
38–21–52–84–86–4

Se derivan los siguientes casos prácticos mediante la selección de los estímulos adecuados basados en el intervalo de distribución que se muestra en la tabla anterior:

HD–P–HD–HD–HD–FZ
P–HD–HD–HD–C–HD–H D
HD–HD–FZ–P–P–HD

El equipo de prueba ejecuta los casos prácticos indicados anteriormente (y otros más) y verifica el comportamiento del software frente a la especificación del sistema. La temporización de las pruebas se registra, de modo que sea posible determinar los intervalos temporales. Mediante el uso de intervalos temporales, el equipo de certificación puede calcular el tiempo-mínimo-entre fallos. Si se lleva a cabo una larga sucesión de pruebas sin fallo, el TMEF es bajo, y se puede suponer que la fiabilidad del software es elevada.

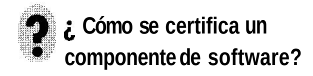
26.4.2. Certificación

Las técnicas de verificación y prueba descritas anteriormente en este capítulo dan lugar a componente5 de software (y a incrementos completos) que se pueden certificar. En el contexto del enfoque de la ingeniería del software de sala limpia, la *certificación* implica que la fiabilidad (medida por el tiempo mínimo de fallo, TMDF) podrá ser especificada para cada componente.

El posible impacto de los componentes de software certificables va más allá de un sencillo proyecto de sala limpia. Los componentes de software reutilizables se pueden almacenar junto con sus escenarios de utilización, con los estímulo\ del programa y con las correspondientes distribuciones de probabilidad. Cada uno de los componentes dispondrá de una fiabilidad certificada dentro del escenario de utilización y dentro del régimen de comprobación descrito. Esta información es sumamente valiosa para otras personas que tengan intención de utilizar estos componentes.

El enfoque de la certificación implica cinco pasos [WOH94]:

1. Es preciso crear escenarios de utilización.
2. Se especifica un perfil de utilización.
3. Se generan casos de prueba a partir del perfil.
4. Se ejecutan pruebas y los datos de los fallos se registran y se analizan.
5. Se calcula y se certifica la fiabilidad.



Los pasos 1 a 4 se han descrito en secciones anteriores. En esta sección, nos concentramos en la certificación de fiabilidad.

[5] Se utilizan herramientas automatizadas con este fin Para mas informacion, vease [DYE92]

La certificación para la ingeniería del software de sala limpia requiere la creación de tres modelos [POO34]:

Modelo de muestreo. La comprobación de software ejecuta *m* casos de prueba aleatorios, y queda certificada si no se produce ningún fallo o si se produce un número de fallos inferior al especificado. El valor de *m* se deriva matemáticamente para asegurar que se alcance la fiabilidad necesaria.

Modelo de componentes. Es preciso certificar un sistema compuesto por *n* componentes. El modelo de componentes capacita al analista para determinar la probabilidad de que falle el componente *i* antes de finalizar el programa.

Modelo de Certificación. Se estima y certifica la fiabilidad global del sistema.

Al final de la prueba estadística de utilización, el equipo de certificación posee la información necesaria para proporcionar un software que tenga un TMEF certificado que se habrá calculado empleando todos estos modelos.

Una descripción detallada del cálculo de los modelos de muestreo, de componentes y de certificación va más allá del alcance de este libro. El lector interesado encontrará detalles adicionales en [MUS87], [CUR86] y [POO93].

RESUMEN

La ingeniería del software de sala limpia es un enfoque formal para el desarrollo del software, que puede dar lugar a un software que posea una calidad notablemente alta. Emplea la especificación de estructura de cajas (o métodos formales) para el modelado de análisis y diseño, y hace hincapié en la verificación de la corrección, más que en las pruebas, como mecanismo fundamental para hallar y eliminar errores. Se aplica una prueba estadística de utilización para desarrollar la información de tasa de fallos necesaria para certificar la fiabilidad del software proporcionado.

El enfoque de sala limpia comienza por unos modelos de análisis y diseño que hacen uso de una representación de estructura de cajas. Una «caja» encapsula el sistema (o algún aspecto del sistema) en un determinado nivel de abstracción. Se utilizan cajas negras para representar el comportamiento observable externamente de ese sistema. Las cajas de estado encapsulan los datos y operaciones de ese estado. Se utiliza una caja limpia para modelar el diseño de procedimientos que está implicado por los datos y operaciones de la caja de estados.

Se aplica la verificación de corrección una vez que está completo el diseño de estructura de cajas. El diseño de procedimientos para un componente de software se desglosará en una serie de subfunciones. Para demostrar la corrección de cada una de estas subfunciones, se definen condiciones de salida para cada una de las subfunciones y se aplica un conjunto de subpruebas. Si se satisfacen todas y cada una de las condiciones de salida, entonces el diseño debe ser correcto.

Una vez finalizada la verificación de corrección, comienza la prueba estadística de utilización. A diferencia de la comprobación condicional, la ingeniería del software de sala limpia no hace hincapié en la prueba unitaria o de integración. En su lugar, el software se comprueba mediante la definición de un conjunto de escenarios, mediante la determinación de las probabilidades de utilización de cada uno de esos escenarios y mediante la aplicación posterior de pruebas aleatorias que satisfagan estas probabilidades. Los registros de error resultantes se combinan con modelos de muestreo, de componentes y de certificación para hacer posible el cálculo matemático de la fiabilidad estimada de ese componente de software.

La filosofía de sala limpia es un enfoque riguroso de la ingeniería del software. Se trata de un modelo de proceso del software que hace hincapié en la verificación matemática de la corrección y en la certificación de la fiabilidad del software. El resultado final son unas tasas de fallo extremadamente bajas, que sería difícil o imposible de conseguir empleando unos métodos menos formales.

REFERENCIAS

[CUR86] Currit, P.A., M. Dyer y H.D. Mills, «Certifying the Reliability of Software», *IEEE Trans. Software Engineering*, vol. SE-12, n." 1, Enero de 1994.

[DYE92] Dyer, **M.**, *The Cleanroom Approach to Quality Software Development*, Wiley, 1992.

[HAU94] Hausler, P.A., R. Linger y C. Trammel, «Adopting Cleanroom Software Engineering with a phased Approach», *IBM Systems Journal*, vol. 33, n.º 1, Enero de 1994, pp. 89-109.

[HEN95] Henderson, J., «Why isn't cleanrooin the Universal Software Development Methodology?», *Crosstalk*, vol. 8, n.º 5, Mayo de 1995, pp. 11-14.

[HEV93] Hevner, A.R., y H.D. Mills, «Box Structure Methods for System Development with Objects», *IBM Systems Journal*, vol. 31, n.º 2, Febrero de 1993, pp. 232-251.

[LIN79] Linger, R. M., H. D. Mills y B.I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, 1979.

[LIN88] Linger, R. M., y H. D. Mills, «A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility», *Proc. COMPSAC '88*, Chicago, Octubre de 1988.

[LIN94] Linger, R., «Cleanroom Process Model», *IEEE Software*, vol. 11, n.º 2, Marzo de 1994, pp. 50-58.

[MIL87] Mills, H.D., M. Dyer y R. Linger, «Cleanroom Software Engineering», *IEEE Software*, vol. 4, n.º 5, Septiembre de 1987, pp. 19-24.

[MIL88] Mills, H.D., «Stepwise Refinement and Verification in Box Structured Systems», *Computer*, vol. 21, n.º 6, Junio de 1988, pp. 23-35.

[MUS87] Musa, J.D., A. Iannino y K. Okumoto, *Engineering und Managing Software with Reliability Measures*, McGraw-Hill, 1987.

[POO88] Poore, J. H., y H.D. Mills, «Bringing Software Under Statistical Quality Control», *Quality Progress*, Noviembre de 1988, pp. 52-55.

[POO93] Poore, J. H., H.D. Mills y D. Mutchler, «Planning and Certifying Software System Reliability», *IEEE Software*, vol. 10, n.º 1, Enero de 1993, pp. 88-99.

[WOH94] Wohlin, C., y P. Runeson, «Certification of Software Components», *IEEE Trans, Software Engineering*, vol. 20, n.º 6, Junio de 1994, pp. 494-499.

PROBLEMAS Y PUNTOS A CONSIDERAR

26.1. Si tuviera que seleccionar un aspecto de la ingeniería del software de sala limpia que la hiciera radicalmente distinta de los enfoques convencionales u orientados a objetos de la ingeniería del software, ¿cuál sería'?

26.2. ¿Cómo se combinan el modelo de proceso incremental y el trabajo de certificación para construir un software de elevada calidad'!

26.3. Empleando la estructura de especificación de cajas, desarrollar un análisis de «primer paso» y unos modelos de diseño para el sistema *HogarSeguro*.

26.4. Desarrolle una especificación de estructura de cajas para una parte del sistema SSRB presentado en el Problema 12.13.

26.5. Desarrolle una especificación de estructura de cajas para el sistema de correo electrónico en el Problema 21.14.

26.6. Se define el algoritmo de ordenación por el método de las burbujas de la forma siguiente:

```
procedure ordenburbuja;
   var i, t : integer;
   begin
   repeat until t = a[1];
      t:=a[1]
      for j := 2 to n do
         if a[j-1] > a[j] then begin
            t:= a[j-1];
            a[j-1]:= a[j]
            a[j] := t;
            end;'
   endrep
enu'
```

Descomponer el diseño en subfunciones y definir un conjunto de condiciones que hagan posible demostrar que este algoritmo es correcto.

26.7. Documentar una demostración de verificación de corrección para la ordenación por el método de la burbuja descrita en el Problema 26.6.

26.8. Seleccionar un componente de programa que se haya diseñado en otro contexto (o bien asignado por el instructor) y desarrollar para él una demostración completa de corrección.

26.9. Seleccionar un programa que se utilice regularmente (por ejemplo, un gestor de correo electrónico, un procesador de texto, una hoja de cálculo). Crear un conjunto de escenarios de utilización para ese programa. Definir la probabilidad de utilización de cada escenario y desarrollar entonces una tabla de estímulos del programa y de distribución de probabilidades parecida a la que se muestra en la Sección 26.4.1.

26.10. Para la tabla de estímulos del programa y de distribución de probabilidades desarrollada en el Problema 26.9, utilizar un generador de números aleatorios con objeto de desarrollar un conjunto de casos de prueba para utilizarlo en una prueba estadística de utilización.

26.11. Con sus propias palabras, describa el objetivo de la certificación en el contexto de la ingeniería del software de sala limpia.

26.12. Escribir un pequeño'trabajo que describa las matemáticas utilizadas para definir los modelos de certificación descritos brevemente en la Sección 26.4.2. Utilícense [MUS87], [CUR86] y [POP93] como punto de partida.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Prowell et al. (*Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, 1999) describe con profundidad todos los aspectos importantes del enfoque de sala limpia. Estudios Útiles sobre los temas de sala limpia han sido editados por Poore y Trammell (*Cleanroom Software: A Reader*, Blackwell Publishing, 1996). Becker y Whittaker (*Cleanroom Software Engineering Practices*, Idea Group Publishing, 1996) presentan una visión general excelente para aquellas personas que no estén familiarizadas con las prácticas de sala limpia.

The Cleanroom Panphlet (Software Technology Support Center, Hill AF Base, UT, abril 1995) contiene reimpresiones de varios artículos importantes. Linger [LIN94] es una de las mejores introducciones a este tema. *Asset Source for Software Engineering Technology*, ASSET (Departamento de Defensa americano) ofrece un conjunto excelente de seis volúmenes

de Cleanroom Engineering Handbooks. Se puede contactar con ASSET en info@source.asset.com. Lockheed Martin ha preparado la guía *Guide to the Integration of Object-Oriented Methods and CleanroomSoftware Engineering* (1997) que contiene un proceso genérico de sala limpia para sistemas operativos y está disponible en:

http://www.asset.com/stars/cleanroom/oo/guidhome.htm

Linger y Trammell *(Cleanroom Software Engineering Reference Model*, SEI Technical Report CMU/SEI-96-TR-022, 1996) han definido un conjunto de 14 procesos de sala limpia y 20 productos de trabajos que forman la base del SEI CMM para la ingeniería del software de sala limpia (CMU/SEI-96-TR-023).

Michael Deck, de Cleanroom Software Engineering, Inc., ha preparado una bibliografía sobre temas de sala limpia. Entre las referencias se encuentran las siguientes:

Generales e Introductorias

Deck, M.D., «Cleanroom Software Engineering Myths and Realities», *Quality Week 1997,* San Francisco, CA, Mayo de 1997.

Deck, M.D, y J.A. Whittaker, «Lessons Learned from Fifteen Years of Cleanroom Testing», *Software Testing, Analysis, and Review (STAR) '97,* San José, CA, 5-9 de Mayo de 1997.

Lokan, C.J., «The Cleanroom for Software Development», *The Australian Computer Journal*, vol. 25, n.º 4, Noviembre de 1993.

Linger, Richard C., «Cleanroom Software Engineering for Zero-Defect Software», *Proc. 15th International Conference Internationul on Software Engineering,* Mayo de 1993.

Keuffel, W., «Clean Your Room: Formal Methods for the 90's», *Computer Language*, Julio de 1992, pp. 39-46.

Hevner, A.R, S.A. Becker y L.B. Pedowitz, «Integrated CASE for Cleanroom Development», *IEEE Software,* Marzo de 1992, pp. 69-76.

Cobb, R.H. y H.D. Mills, «Engineering Software under Statistical Quality Control», *IEEE Software*, Noviembre de 1990, pp. 44-45.

Practicas de Gestión

Becker, S.A., M.D. Deck y T. Janzon, «Cleanroom and Organizational Change», *Proc. 14th Pacific Northwest Software Quality Conference*, Portland, OR, 29-30 de Octubre de 1996.

Linger, R.C., «Cleanroom Process Model», *IEEE Software,* marzo de 1994, pp. 50-58.

Linger, R.C. y Spangler, R.A., «The IBM Cleanroom Software Engineering Technology Transfer Program», *Sixth SEI Conference on Software Engineering Education,* San Diego, CA, Octubre de 1992.

Especificación, Diseño y Revisión

Deck, M.D., «Cleanroom and Object-Oriented Software Engineering: A Unique Synergy», *I Y96 Software Technology Conference,* Salt Lake City, UT, 24 de Abril de 1996.

Deck, M.D., «Using Box Structures to Link Cleanroom and Object-Oriented Software Engineering», *Technical Report 94.01h*, Cleanroom Software Engineering Inc., Boulder, CO, 1994.

Dyer, M. «Designing Software for Provable Correctness: the direction for quality software», *Information and Software Technology*, vol. *30*, n.º 6, Julio/Agosto de 1988, pp. 331-340.

Pruebas y Certificación

Dyer, M., «An Approach to Software Reliability Measurement», *Information and Software Technology,* vol. 29, n.º 8, Octubre de 1987, pp. 415-420.

Head, G.E., «Six-Sigma Software Using Cleanroom Software Engineering Techniques», *Hewlett-Packard Journal*, Junio de 1994, pp. 40-50.

Oshana, R., «Quality Software via a Cleanroom Methodology», *Emhedded Systems Programming*, Septiembre de 1996, pp. 36-52.

Whittaker, J.A., y Thomason, M.G., «A Markov Chain Model for Statistical Software Testing», *IEEE Trans. on Software Engineering,* Octubre de 1994, pp. 812-824.

Estudios de casos e informes experimentales

Head, G.E., «Six-Sigma Software Using Cleanrooin Software Engineering Techniques», Hewlett-Packard Journal, Junio de 1994, pp. 40-SO.

Hevner, A.R., y H.D. Mills, «Box-structured methods for systems development with objects», *IBM systems Journal*, vol. *32*, n.º 2, 1993, pp. 232-251.

Tann, L-G., «OS32 and Cleanroom», *Proc. 1st Annual European Industrial Symposium on Cleanroom Software Engineering,* Copenhagen, Denmark, 1993, pp. 1-40.

Hausler, P.A., «A Recent Cleanroom Success Story: The Redwing Project», *Proc. 17th Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, Diciembre de 1992.

Trammel, C.J., Binder L.H. y Snyder, C.E., «The Automated Production Control Documentation System: A Case Study in Cleanroom Software Engineering», *ACM Trans. on Software Engineering and Methodology*, vol. 1, n.º 1, Enero de 1992, pp. 81-94.

La verificación de diseños mediante pruebas de corrección se encuentra en el centro del enfoque de sala limpia. En los libros de Baber *(Error-Free Software,* Wiley, 1991) y Schulmeyer (*Zero Deject Software,* McGraw-Hill, 1990) se estudia la prueba de corrección de forma muy detallada.

En Internet se puede encontrar disponible mucha información variada sobre la ingeniería del software dc sala limpia y sobre temas relacionados. Para conseguir una lista actualizada de referencias que sea relevante para la ingeniería del software de sala limpia se puede visitar **http://www pressman5.com**

.

CAPÍTULO

27 INGENIERÍA DEL SOFTWARE BASADA EN COMPONENTES

EN el contexto de la ingeniería del software, la reutilización se puede considerar una idea nueva y antigua. Los programadores han reutilizado ideas, abstracciones y procesos desde el principio de la era de los computadores, pero el primer enfoque de reutilización era muy concreto. Hoy en día, los sistemas complejos y de alta calidad basados en computadora se deben construir en períodos de tiempo muy cortos. Esto se mitiga con un enfoque de reutilización más organizado.

La *ingeniería del software basada en componentes* (ISBC) es un proceso que se centra en el diseño y construcción de sistemas basados en computadora que utilizan «componentes» de software reutilizables. Clements [CLE95] describe la ISBC de la manera siguiente:

> [La ISBC] está cambiando la forma en que se desarrollan los sistemas de software. [La ISBC] representa la filosofía de «comprar, no construir», que expusieron Fred Brooks y otros. De la misma manera que las primeras subrutinas liberaban al programador de tener que pensar en detalles, [ISBC] cambia su objetivo y pasa de programar el software a componer sistemas de software. La implementación ha dado paso a la integración como núcleo del enfoque. Se puede decir que en su base se encuentra la suposición de que en muchos sistemas grandes de software existe una base común suficiente como para justificar los componentes reutilizables para explotar y satisfacer a esa base común.

Sin embargo, surgen muchas preguntas. ¿Es posible construir sistemas complejos ensamblándolos a partir de un catálogo de componentes de software reutilizables? ¿Se puede conseguir de una manera rentable y en poco tiempo? ¿Se pueden establecer incentivos para animar a que los ingenieros del software reutilicen y no reinventen? ¿Están dispuestos los directivos a contraer los gastos

VISTAZO RÁPIDO

¿Qué es? Compre un equipo de música estéreo y lléveselo a casa. Cada componente ha sido diseñado para acoplarse en un estilo arquitectónico específico —las conexiones son estándar y puede preestablecerse el protocolo de comunicación—. El ensamblaje es fácil porque el sistema no tiene que construirse a partir de piezas por separado. La ingeniería del software basada en componentes (ISBC) lucha por conseguir lo mismo. Un conjunto de componentes de software preconstruidos y estandarizados están disponibles para encajar en un estilo arquitectónico específico pura algún dominio de aplicación. La aplicación se ensambla entonces utilizando estos componentes y no las .piezas por separado. de un lenguaje de programación convencional.

¿Quién lo hace? Los ingenieros del software.

¿Por qué es importante? La instalación del equipo estéreo solo lleva unos pocos minutos, porque los componentes están diseñados para integrarse con facilidad. Aunque el software es considerablemente más complejo que el sistema estéreo, se puede seguir diciendo que los sistemas basados en componentes son más fáciles de ensamblar y, por tanto, más caros de construir a partir de piezas separadas. Además, la ISBC hace hincapié en la utilización de patrones arquitectónicos predecibles y en una infraestructura de software estándar, lo que lleva a un resultado de calidad superior.

¿Cuáles son los pasos? La ISBC acompaña a dos actividades de ingeniería paralelas: la ingeniería del dominio y el desarrollo basado en componentes. La ingeniería del dominio explora un dominio de aplicaciones con la intención de encontrar específicamente los componentes de datos funcionales y de comportamiento candidatos para la reutilización. Estos componentes se encuentran en bibliotecas de reutilización. El desarrollo basado en componentes obtiene los requisitos del cliente y selecciona el estilo arquitectónico adecuado para cumplir los objetivos del sistema que se va a construir, y a continuación: (1) selecciona posibles componentes para la reutilización; (2) cualifica los componentes para asegurarse de que encajan adecuadamente en la arquitectura del sistema; (3) adapta los componentes si se deben hacer modificaciones para poderlos integrar adecuadamente; (4) integra los componentes para formar subsistemas y la aplicación completa. Además, los componentes personalizados se han diseñado para afrontar esos aspectos del sistema que no pueden implementarse utilizando componentes que ya existen.

¿Cuál es el producto obtenido? El producto de la ISBC es el software operacional ensamblado utilizando los componentes de software existentes y los que se acaban de desarrollar.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Utilizando las mismas prácticas SQA que se aplican en todos los procesos de ingeniería del software —las revisiones técnicas formales evalúan los modelos de análisis y diseño—: las revisiones especializadas tienen en consideración temas asociados con los componentes adquiridos; y la comprobación se aplica para descubrir errores en el software nuevo y en componentes reutilizables que se hayan integrado en la arquitectura.



añadidos y asociados con la creación de componentes de software reutilizables? ¿Se puede crear una biblioteca con los componentes necesarios para llevar a cabo la reutilización de manera accesible para aquellos que la necesitan?

Estas y otras preguntas siguen vivos en la comunidad de investigadores y de profesionales de la industria que luchan por hacer que la reutilización de componentes de software sea el enfoque más convencional de la ingeniería del software. Algunas de estas preguntas se estudian en este capítulo.

27.1 INGENIERÍA DE SISTEMAS BASADA EN COMPONENTES

Superficialmente, la ISBC parece bastante similar a la ingeniería del software orientada a objetos. El proceso comienza cuando un equipo de software establece los requisitos del sistema que se va a construir utilizando las técnicas convencionales de obtención de requisitos (Capítulos 10 y 11). Se establece un diseño arquitectónico (Capítulo 14), pero en lugar de entrar inmediatamente en tareas de diseño detalladas, el equipo examina los requisitos para determinar cuál es el subsistema que está dispuesto para la *composición*, y no para la construcción. Esto es, el equipo formula las siguientes preguntas para todos y cada uno de los requisitos del sistema:

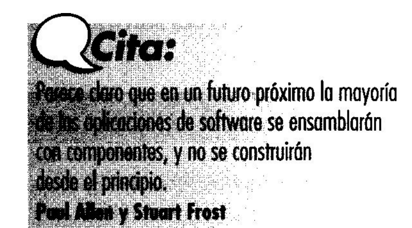
- ¿Es posible disponer de componentes comerciales ya desarrollados (CYD) para implementar el requisito?
- ¿Se dispone de componentes reutilizables desarrollados internamente para implementar el requisito?
- ¿Son compatibles las interfaces de los componentes que están disponibles dentro de la arquitectura del sistema a construir?

El equipo intenta modificar o eliminar aquellos requisitos del sistema que no se pueden implementar con componentes CYD o de desarrollo propio[1]. Si los requisitos no se pueden ni cambiar ni borrar, se aplican métodos de ingeniería del software convencionales u orientados a objetos para desarrollar los componentes nuevos que se deben diseñar para cumplir los requisitos. Pero para esos requisitos que se afrontan con los componentes disponibles comienza un conjunto diferente de actividades de ingeniería del software:

Cualificación de componentes. Los requisitos del sistema y la arquitectura definen los componentes que se van a necesitar. Los componentes reutilizables (tanto si son CYD como de desarrollo propio) se identifican normalmente mediante las características de sus interfaces. Es decir, se describen «los servicios que se proporcionan y el medio por el que los consumidores acceden a estos servicios» [BRO96] como parte de la interfaz del componente. Pero la interfaz no proporciona una imagen completa del acople del componente en la arquitectura y en los requisitos. El ingeniero del software debe de utilizar un proceso de descubrimiento y de análisis para cualificar el ajuste de cada componente.



Adaptación de componentes. En el Capítulo 14 se señaló que la arquitectura del software representa los patrones de diseño que están compuestos de componentes (unidades de funcionalidad), conexiones y coordinación. Esencialmente la arquitectura define las normas del diseño de todos los componentes, identificando los modos de conexión y coordinación. En algunos casos, es posible que los componentes reutilizables actuales no se correspondan con las normas del diseño de la arquitectura. Estos componentes deben de adaptarse para cumplir las necesidades de la arquitectura o descartarse y reemplazarse por otros componentes más adecuados.



Composición de componentes. El estilo arquitectónico vuelve a jugar un papel clave en la forma en que los componentes del software se integran para formar un sistema de trabajo. Mediante la identificación de los mecanismos de conexión y coordinación (por ejemplo, las propiedades de ejecución en el diseño), la arquitectura dicta la composición del producto final.

Actualización de componentes. Cuando se implementan sistemas con componentes CYD, la actualización se complica por la imposición de una tercera parte (es decir, es posible que la empresa que desarrolló el componente reutilizable no tenga el control de la empresa de ingeniería del software).

Todas y cada una de las actividades ISBC se estudian más profundamente en la Sección 27.4.

En la primera parte de esta sección el término «componente» se ha utilizando en repetidas ocasiones, a pesar de que es difícil de efectuar una descripción definitiva del término. Brown y Wallnau [BRO96] sugieren las siguientes posibilidades:

[1] La implicación es que la organización ajusta sus requisitos comerciales o del producto de manera que se puede lograr la implementación basada en componentes sin la necesidad de una ingeniería personalizada. Este enfoque reduce el coste del sistema y mejora el tiempo de comercialización, pero no siempre es posible.

- *componente*– una parte reemplazable, casi independiente y no trivial de un sistema que cumple una función clara en el contexto de una arquitectura bien definida;
- *componente del software en ejecución*– un paquete dinámico de unión de uno o más programas gestionados como una unidad, a los que se accede a través de interfaces documentadas que se pueden descubrir en la ejecución;
- *componente de software*– una unidad de composición que solo depende del contexto contractual de forma específica y explícita;
- *componente de negocio*– la implementación de software de un concepto comercial «autónomo» o de un proceso comercial;

Además de las descripciones anteriores, los componentes del software también se pueden caracterizar por el uso en el proceso ISBC. Además de los componentesCYD, el proceso ISBC produce los siguientes componentes:

- *componentes cualificados*– evaluados por los ingenieros de software para asegurar que no sólo la funcionalidad sino también el rendimiento, la fiabilidad y otros factores de calidad (Capítulo 19) encajan con los requisitos del sistema/producto que se va a construir;

Referencia cruzada

La certificación de componentes se puede realizar con los métodos de sala limpio. Para obtener más información consulte el Capítulo 26.

- *componentes adaptados*– adaptados para modificar (también llamados «enmascarados» o «envoltorios») [BRO96] las características no deseadas.
- *componentes ensamblados*– integrados en un estilo arquitectónico e interconectados con una infraestructura de componentes adecuada que permite coordinar y gestionar los componentes de forma eficaz;
- *componentes actualizados*– el software actual se reemplaza a medida que se dispone de nuevas versiones de componentes;

Dado que la ISBC es una disciplina en evolución, no es probable que en el futuro surja una definición unificada.

27.2 EL PROCESO DE ISBC

En el Capítulo 2, se utilizó un «modelo de desarrollo basado en componentes» (Fig. 2.12) para ilustrar la forma en que se integra una biblioteca de «componentes candidatos» reutilizables en un modelo típico de proceso evolutivo. Sin embargo, el proceso ISBC se debe caracterizar de forma que no sólo identifique los componentes candidatos sino que también cualifique la interfaz de cada componente, que adapte los componentes para extraer las faltas de coincidencias arquitectónicas, que ensamble los componentes en un estilo arquitectónico seleccionado y que actualice los componentes a medida que cambian los requisitos del sistema [BRO96].

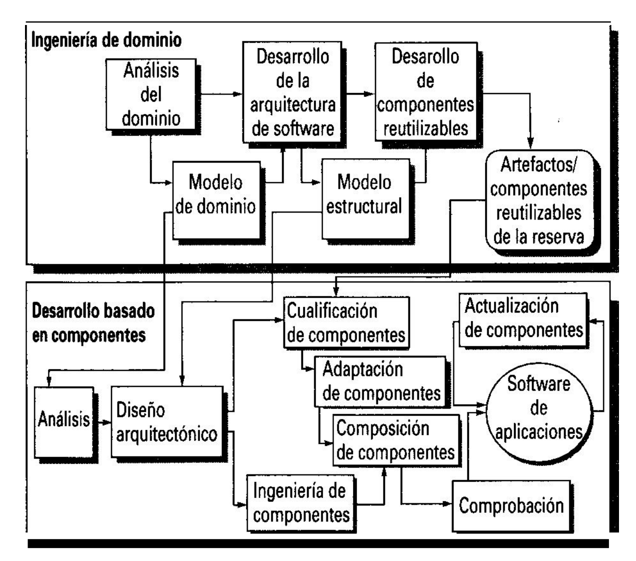


FIGURA 27.1. Un modelo de proceso de soporte a la ISBC.

El modelo de proceso para la ingeniería del software basada en componentes hace hincapié en las pistas paralelas en las que aparece concurrentemente la ingeniería del dominio (Sección 27.3) con el desarrollo basado en componentes. La *ingeniería del dominio* realiza el trabajo que se requiere para establecer el conjunto de componentes de software que el ingeniero del software puede reutilizar. Estos componentes entonces se transfieren a través de un «límite» que separa la ingeniería del dominio del desarrollo basado en componentes.



La página de tecnología de componentes proporciona información útil sobre ISBC:
www.odateam.com/cop/

La Figura 27.1 ilustra un modelo de proceso típico que acopla la ISBC explícitamente [CHR95]. La ingeniería del dominio crea un modelo de dominio de aplicación que se utiliza como base para analizar los requisitos del usuario en el flujo de la ingeniería del software. Una arquitectura de software genérica (y los puntos de estructura correspondientes, véase la Sección 27.3.3) proporciona la entrada para el diseño de la aplicación. Finalmente, después de que se han comprado los componentes reutilizables, se han seleccionado a partir de las bibliotecas existentes o se han construido (como parte de la ingeniería del dominio), los ingenieros del software dispondrán de ellos durante la actividad de desarrollo basada en componentes.

27.3 INGENIERÍA DEL DOMINIO

El objetivo de la ingeniería del dominio es identificar, construir, catalogar y diseminar un conjunto de componentes de software que tienen aplicación en el software actual y futuro dentro de un dominio de aplicación en particular. El objetivo general consiste en establecer mecanismos que capaciten a los ingenieros del software para compartir estos componentes —para reutilizarlos— a lo largo de su trabajo en sistemas nuevos o actuales.

Cita:

La ingeniería de dominios está a punto de encontrar referencias en sistemas para identificar los componentes que se pueden aplicar a muchos sistemas, y para identificar las familias de programas que están ubicadas de forma que saquen provecho de esos componentes.

Paul Clements

La ingeniería del dominio incluye tres actividades principales: análisis, construcción y diseminación. En el Capítulo 21 se presentó una revisión general del análisis del dominio. Sin embargo, este tema se vuelve a tratar en las secciones siguientes. La construcción y la diseminación del dominio se describirán más adelante en otras secciones dentro de este mismo capítulo.

Se podría argumentar que «la reutilización desaparecerá, no mediante la eliminación, sino mediante la integración» en el entramado de prácticas de la ingeniería del software [TRA95]. Como la reutilización cada vez recibe más importancia, todavía hay quien cree que en la próxima década la ingeniería del dominio tendrá tanta importancia como la ingeniería del software.

27.3.1. El proceso de análisis del dominio

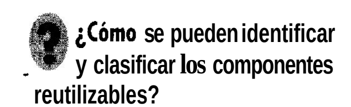
En el Capítulo 21 se describía el enfoque general del análisis del dominio en el contexto de la ingeniería del software orientado a objetos. Las pasos del proceso se definían de la siguiente manera:

1. Definir el dominio que hay que investigar.
2. Categorizar los elementos extraídos del dominio.
3. Recoger una muestra representativa de las aplicaciones del dominio.
4. Analizar cada aplicación de la muestra.
5. Desarrollar un modelo de análisis para los objetos.

Es importante tener en cuenta que el análisis del dominio se puede aplicar a cualquier paradigma de la ingeniería del software, siendo posible aplicarlo tanto para el desarrollo convencional como para el desarrollo orientado a objetos.

Prieto-Diaz [PRI87] amplía el segundo paso del análisis del dominio indicado anteriormente y sugiere un enfoque de ocho pasos para la identificación y clasificación de componentes reutilizables:

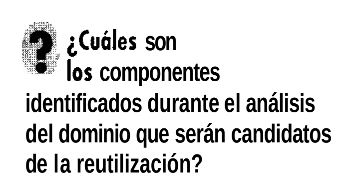
1. Seleccionar funciones y objetos específicos.
2. Abstraer funciones y objetos.
3. Definir una taxonomía.
4. Identificar las características comunes.
5. Identificar las relaciones específicas.
6. Abstraer las relaciones.
7. Derivar un modelo funcional.
8. Definir un lenguaje del dominio.


¿Cómo se pueden identificar y clasificar los componentes reutilizables?

El *lenguaje del dominio* hace posible la especificación y construcción posterior de aplicaciones dentro del dominio.

Aun cuando los pasos indicados anteriormente proporcionan un modelo útil para el análisis del dominio, no proporcionan ninguna guía para decidir cuáles son los componentes de software que son candidatos para la reutilización. Hutchinson y Hindley [HUT88] sugieren el siguiente conjunto de cuestiones pragmáticas como guía para identificar los componentes del software reutilizables:

- ¿Es la funcionalidad del componente un requisito para futuras implementaciones ?
- ¿Hasta qué punto es corriente la función del componente dentro del dominio?
- ¿Existe una duplicación de la función del componente dentro del dominio?
- ¿Depende ese componente del hardware?
- ¿Permanece intacto el hardware entre implementaciones?


¿Cuáles son los componentes identificados durante el análisis del dominio que serán candidatos de la reutilización?

- ¿Es posible trasladar las partes específicas del hardware a otro componente?
- ¿Está el diseño suficientemente optimizado para la siguiente implementación?
- ¿Es posible parametrizar un componente no reutilizable para que pase a ser reutilizable?
- ¿Es reutilizable ese componente en muchas implementaciones con cambios solo menores?
- ¿Es viable la reutilización mediante modificaciones?
- ¿Se puede descomponer un componente no reutilizable para producir componentes reutilizables?
- ¿Hasta qué punto es válida la descomposición del componente para su reutilización?

Una descripción en profundidad de los métodos de análisis del dominio va más allá del alcance de este libro. Para más información acerca del análisis del dominio, véase [PRI93].

27.3.2. Funciones de caracterización

A veces resulta difícil determinar si un componente potencialmente reutilizable es realmente aplicable en una situación determinada. Para llevar a cabo esta determinación, es necesario definir un conjunto de características del dominio que sean compartidas por todo el software en el seno del dominio. Una característica del dominio define algún atributo genérico de todos los productos que existen dentro del dominio. Por ejemplo, entre las características genéricas se podría incluir: la importancia de la seguridad y fiabilidad, el lenguaje de programación, la concurrencia de procesamiento, y muchas más.

Un conjunto de características de dominio de un componente reutilizable se puede representar como $\{D_p\}$ en donde cada elemento D_{pi} del conjunto representa una característica específica de dominio. El valor asignado a D_{pi} representa una escala ordinal como una indicación de la relevancia de esa característica para el componente *p*. Una escala típica [BAS94] podría ser:

1. no es relevante para que la reutilización sea o no adecuada.
2. relevante sólo en circunstancias poco comunes.
3. relevante: el componente se puede modificar para poder utilizarlo, a pesar de las diferencias.
4. claramente relevante, y si el software nuevo no posee esta característica, la reutilización será ineficiente pero quizás siga siendo posible
5. es claramente relevante, y si el software nuevo no posee esta característica, la reutilización será ineficiente y la reutilización sin esta característica no es recomendable.

Producto	Proceso	Personal
Estabilidad de requisitos	Modelo de proceso	Motivación
Software concurrente	Ajuste del proceso	Educación
Restricciones de memoria	Entorno del proyecto	Experiencia/ formación
Tamaño de aplicaciones	Restricciones de planificación	• dominio de aplicación
Complejidad de interfaz de usuario	Restricciones de presupuesto	• proceso
Lenguaje(s) de programación	Productividad	• plataforma
Seguridad y fiabilidad		• lenguaje
Requisitos de la duración global		Productividad del equipo de desarrollo
Calidad del producto		
Fiabilidad del producto		

TABLA 27.1. Características del dominio que afectan al software [BAS94].

Cuando se va a construir un software nuevo, *w*, dentro del dominio de la aplicación, se deriva para él un conjunto de características del dominio. A continuación, se efectúa una comparación entre D_{pi} y D_{wi}, para determinar si el componente p se puede reutilizar eficazmente en la aplicación *w*.



Referencia Web

Para encontrar referencias valiosas sobre una tecnología de objetos de direccionamiento de informes, arquitecturas y análisis de dominios, se puede visitar la dirección de Internet:
www.sei.cmu.edu/mbse/ wisr_report.html

La Tabla 27.1 [BAS94] enumera las características típicas del dominio que pueden tener impacto en la reutilización del software. Estas características del dominio deben de tenerse en cuenta con objeto de reutilizar un componente de forma eficiente.

Aun cuando el software que se vaya a construir exista claramente en el seno de un dominio de aplicación, los componentes reutilizables situados dentro de ese dominio deberán ser analizados con objeto de determinar su aplicabilidad. En algunos casos (esperemos que sea un número limitado), la «reinvención de la rueda» puede seguir siendo la opción más rentable.

27.3.3. Modelado estructural y puntos de estructura

Cuando se aplica el análisis del dominio, el analista busca tramas repetidas en las aplicaciones que residen dentro del dominio. El *modelado **estructural*** es un enfoque de ingeniería basado en tramas que opera efectuando la suposición consistente en que todo dominio de aplicación posee tramas repetidas (de función, de datos y de comportamiento) que tienen un potencial de reutilización.

Pollak y Rissman [POL94] describen los modelos estructurales de la siguiente manera:

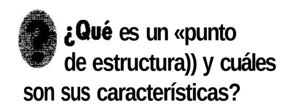
> **Los modelos estructurales constan de un pequeño número de elementos estructurales que manifiestan unas tramas de interacción claras. Las arquitecturas de sistemas que utilizan modelos estructurales se caracterizan por múltiples ensamblajes formados por estos elementos del modelo. De esta manera, las interacciones complejas entre sistemas que tienen muchas unidades arquitectónicas, surgen de tramas sencillas de interacción que existen en el conjunto pequeño de elementos.**

Todo dominio de aplicación se puede caracterizar por un modelo estructural (por ejemplo, la aviónica difiere mucho en los aspectos específicos, pero todo el soft-

ware moderno de este dominio tiene el mismo modelo estructural). Por tanto, el modelo estructural es un estilo arquitectónico (Capítulo 14) que puede y debe reutilizarse en aplicaciones pertenecientes al dominio.

McMahon [MCM95] describe un *punto de estructura* como una «estructura bien diferenciada dentro de un modelo estructural». Los puntos de estructura tienen tres características bien claras:

1. Un punto de estructura es una abstracción que debe de tener un número limitado de instancias. Al replantear esta característica en la jerga orientada a objetos (Capítulo 20), el tamaño de la jerarquía de clases debe ser pequeño. Además, la abstracción debe repetirse a lo largo de las aplicaciones del dominio. En caso contrario, el esfuerzo requerido para verificar, documentar y diseminar ese punto de estructura no puede justificarse en términos de coste.


¿Qué es un «punto de estructura)) y cuáles son sus características?

2. La reglas que rigen la utilización del punto de estructura deben entenderse fácilmente. Además, la interfaz con el punto de estructura debe ser relativamente sencilla.
3. El punto de estructura debe implementar el ocultamiento de información aislando toda la complejidad dentro del punto de estructura en **sí.** Esto reduce la complejidad percibida del sistema en su conjunto.

Como ejemplo de puntos de estructura como tramas arquitectónicas para un sistema, considérese el dominio del software de sistemas de alarma. El dominio del sistema puede abarcar sistemas tan simples como *HogarSeguro* (descritos en capítulos anteriores) o tan complejos como el sistema de alarma de un proceso industrial. Sin embargo, se encuentran un conjunto de tramas estructurales predecibles:

- una *interfaz* que capacita al usuario para interactuar con el sistema;
- un *mecanismo de establecimiento de límites* que permite al usuario establecer límites sobre los parámetros que hay que medir;
- un *mecanismo de gestión* de sensores que se comunica con todos los sensores empleados en la monitorización;
- un *mecanismo de respuesta* que reacciona frente a las entradas proporcionadas por el sistema de gestión de sensores;
- un *mecanismo de control* que capacita al usuario para controlar la forma en que se efectúa la monitorización.


CLAVE

Un punto de estructura se puede ver como un patrón de diseño que aparece repetidas veces en aplicaciones dentro de un dominio específico.

Cada uno de estos puntos de estructura está integrado en una arquitectura de dominio.

Es posible definir los puntos de estructura genéricos que trascienden a un número de dominios de aplicaciones diferentes [STA94]:

Aplicaciones cliente. La *IGU*, que incluye todos los mentís, paneles y capacidades de entrada y edición de órdenes.

Bases de datos. El depósito de todos los objetos relevantes para el dominio de la aplicación

Motores de cálculo. Los modelos numéricos y no numéricos que manipulan datos.

Función de reproducción de informes. La función que produce salidas de todo tipo.

Editor de aplicaciones. Mecanismo adecuado para personalizar la aplicación ajustándola a las necesidades de usuarios específicos.

Los puntos de estructura se han sugerido como alternativa a las líneas de código y a los puntos de función para la estimación del coste del software ([MCM95]. En la Sección 27.6.2 se presenta una descripción breve del coste utilizando puntos de estructura.

27.4 DESARROLLO BASADO EN COMPONENTES

El desarrollo basado en componentes es una actividad de la ISBC que tiene lugar en paralelo a la ingeniería del dominio. La utilización de métodos de diseño arquitectónico y de análisis se ha descrito anteriormente en otra sección de este texto, en donde el equipo del software refina el estilo arquitectónico adecuado para el modelo de análisis de la aplicación que se va a construir[2].

Una vez que se ha establecido la arquitectura, se debe popularizar mediante los componentes que (1) están disponibles en bibliotecas de reutilización, y/o (2) se han diseñado para satisfacer las necesidades del cliente. De aquí que el flujo de una tarea de desarrollo basada en componentes tenga dos caminos posibles (Fig. 27.1). Cuando los componentes reutilizables están disponibles para una integración futura en la arquitectura, deben

[2] Se debería destacar que el estilo arquitectónico suele estar influenciado por el modelo de estructura genérico creado en la ingeniería del dominio (véase Figura 27.1)

estar cualificados y adaptados. Cuando se requieren componentes nuevos, deben diseñarse. Los componentes resultantes entonces se «componen» (se integran) en una plantilla de arquitectura y se comprueban a conciencia.

27.4.1. Cualificación, adaptación y composición de componentes

Como se acaba de ver, la ingeniería del dominio proporciona la biblioteca de componentes reutilizables necesarios para la ingeniería del software basada en componentes. Algunos de estos componentes reutilizables se desarrollan dentro de ella misma, otros se pueden extraer de las aplicaciones actuales y aun otros se pueden adquirir de terceras partes.

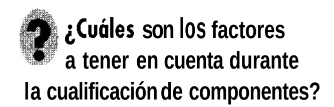
Desgraciadamente, la existencia de componentes reutilizables no garantiza que estos componentes puedan integrarse fácilmente, o de forma eficaz, en la arquitectura elegida para una aplicación nueva. Esta es la razón por la que se aplica una sucesión de actividades de desarrollo basada en componentes cuando se ha propuesto que se utilice un componente.

Cualificación de componentes

La *cualificación de componentes* asegura que un componente candidato llevará a cabo la función necesaria, encajará además «adecuadamente» en el estilo arquitectónico especificado para el sistema y exhibirá las características de calidad (por ejemplo, rendimiento, fiabilidad, usabilidad) necesarias para la aplicación.

La descripción de la interfaz proporciona información útil sobre la operación y utilización de los componentes del software, pero no proporciona toda la información necesaria para determinar si un componente propuesto puede de hecho volver a reutilizarse de manera eficaz en una aplicación nueva. A continuación, se muestran algunos factores de los muchos a tener en cuenta durante la cualificación de los componentes [BRO96]:

- la interfaz de programación de aplicaciones (API);
- las herramientas de desarrollo e integración necesarias para el componente;
- requisitos de ejecución, entre los que se incluyen la utilización de recursos (por ejemplo, memoria o almacenamiento), tiempo o velocidad y protocolo de red;

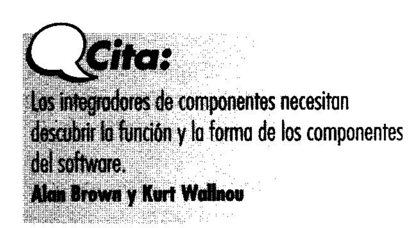


- requisitos de servicio, donde se incluyen las interfaces del sistema operativo y el soporte por parte de otros componentes;
- funciones de seguridad, como controles de acceso y protocolo de autenticación;
- supuestos de diseños embebidos, incluyendo la utilización de algoritmos numéricos y no numéricos;
- manipulación de excepciones.

Cada uno de los factores anteriores es relativamente fácil de valorar cuando se proponen los componentes reutilizables que se han desarrollado dentro de la misma aplicación. Si se han aplicado prácticas de ingeniería del software de buena calidad durante el desarrollo, se pueden diseñar respuestas para las preguntas relacionadas con la lista anterior. Sin embargo, es mucho más difícil determinar el funcionamiento interno de componentes CYD o de terceras partes, porque la Única información disponible es posible que sea la misma interfaz de especificaciones.

Adaptación de componentes

Lo ideal sería que la ingeniería del dominio creara una biblioteca de componentes que pudieran integrarse fácilmente en una arquitectura de aplicaciones. La implicación de una «integración fácil» es que: (1) se hayan implementado los métodos consecuentes de la gestión de recursos para todos los componentes de la biblioteca; (2) que existan actividades comunes, tales como la gestión de datos para todos los componentes, y (3) que se hayan implementado interfaces dentro de la arquitectura y con el entorno externo de manera consecuente.



En realidad, incluso después de haber cualificado un componente para su utilización dentro de una arquitectura de aplicación, es posible exhibir conflictos en una o más de las áreas anteriores. Para mitigar estos conflictos se suele utilizar una técnica de adaptación llamada «encubrimiento de componentes» [BRO96]. Cuando un equipo de software tiene total acceso al diseño interno y al código de un componente (no suele ser el caso de los componentes CYD) se aplica el encubrimiento de caja blanca. Al igual que su homólogo en las pruebas del software (Capítulo 17), el *encubrimiento de caja blanca* examina los detalles del procesamiento interno del componente y realiza las modificaciones a nivel de código para eliminar los conflictos. El *encubrimiento de caja gris* se aplica cuando la biblioteca de componentes proporciona un lenguaje de extensión de componentes, o API, que hace posible eliminar o enmascarar los conflictos. El *encubrimiento de caja negra* requiere la introducción de un preprocesamiento o postprocesamiento en la interfaz de componentes para eliminar o enmascarar conflictos. El equipo de software debe determinar si se justifica el esfuerzo requerido para envolver adecuadamente un componente o si por el contrario se debería diseñar un componente personalizado (diseñado para eliminar los conflictos que se encuentren).

Composición de componentes

La tarea de *composición de componentes* ha ensamblado componentes cualificados, adaptados y diseñados para popularizar la arquitectura establecida para una aplicación. Para poder llevarlo a cabo, debe establecerse una infraestructura donde los componentes estén unidos a un sistema operacional. La infraestructura (normalmente una biblioteca de componentes especializados) proporciona un modelo para la coordinación de componentes y servicios específicos que hacen posible coordinar unos componentes con otros, y llevar a cabo las tareas comunes.

Entre los muchos mecanismos para crear una infraestructura eficaz, existe un conjunto de cuatro «ingredientes arquitectónicos» [ADL95] que se debería presentar para lograr la composición de componentes:

Modelo de intercambio de datos. Se trata de mecanismos que capacitan a los usuarios y a las aplicaciones para interactuar y transferir datos (por ejemplo, arrastrar y soltar, cortar y pegar), y que deberían estar definidos para todos los componentes reutilizables. Los mecanismos de intercambio de datos no solamente permiten la transferencia de datos entre hombre y programa, y entre componentes, **sino** que también hacen posible la transferencia entre los recursos del sistema (por ejemplo, arrastrar un archivo al icono de una impresora para imprimirlo).

Automatización. Para facilitar la interacción entre componentes reutilizables se deberían de implementar herramientas, macros y guiones.

¿Cuáles son los ingredientes necesarios para lograr la composición de componentes?

Almacenamiento estructurado. Los datos heterogéneos (por ejemplo, los datos gráficos, de voz, texto, vídeo y datos numéricos) dentro de un «documento compuesto», deben estar organizados para poder acceder a ellos como si de una sola estructura de datos se tratase, en lugar de comportarse como si fueran toda una colección de archivos por separado. «Los datos estructurados mantienen un índice descriptivo de estructuras de anidamiento, que las aplicaciones pueden recorrer libremente para localizar, crear o editar el contenido de datos individuales según lo indique el usuario final» [ADL95].

Modelo de objetos subyacente. El modelo de objetos asegura que los componentes desarrollados en distintos lenguajes de programación que residan en distintas plataformas pueden ser interoperables. Esto es, los objetos deben ser capaces de comunicarse en una red. Para lograr esto, el modelo de objetos define un estándar para la interoperabilidad de los componentes.

Dado que el impacto futuro de la reutilización y de la ISBC en la industria del software es enorme, una gran cantidad de empresas importantes y consorcios industriales[3] han propuesto estándares para el software por componentes:

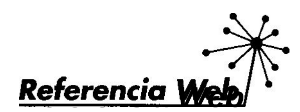
OMG/CORBA. El Grupo de gestión de objetos (Object Management Group) ha publicado una *arquitectura común de distribución de objetos* (OMG/CORBA). Los distribuidores de objetos (ORB) proporcionan toda una gama de servicios que hacen posible que los componentes reutilizables (objetos) se comuniquen con otros componentes, independientemente de su ubicación dentro del sistema. Cuando se construyen componentes empleando el estándar OMG/CORBA, la integración de esos componentes (sin modificación) dentro del sistema queda garantizada si se crea una interfaz mediante un *lenguaje de definición de interfaces* (LDI) para cada componente.



La información más actualizada sobre CORBA se puede obtener en **www.omg.org**

Utilizando una metáfora cliente/servidor, los objetos situados dentro de la aplicación cliente solicitan uno o más servicios del servidor de ORB. Las solicitudes se hacen a través del LDI, o bien dinámicamente en el momento de la ejecución. Un repositorio de interfaces contiene toda la información necesaria acerca de las solicitudes de servicio y de los formatos de respuesta. CORBA se estudia con más detalle en el Capítulo **28.**

COM de Microsoft. Microsoft ha desarrollado un modelo de objetos para componentes (COM) que proporciona una especificación para utilizar los componentes elaborados por diferentes fabricantes dentro de una aplicación única bajo el sistema operativo Windows. COM abarca dos elementos: las interfaces COM (implementadas como objetos COM) y un conjunto de mecanismos para registrar y pasar mensajes entre interfaces COM. Desde el punto de vista de la aplicación, «el foco no está en cómo [se implementan los objetos COM], sino en el hecho de que el objeto tiene una interfaz que se registra con el sistema, y que utiliza el sistema de componentes para comunicarse con otros objetos COM» [HAR98].



La información más actualizada sobre COM se puede obtener en **www.microsoft.com/COM**

Componentes JavaBean de SUN. El sistema de componentes JavaBean es una infraestructura ISBC portátil e independiente de la plataforma que utiliza el lenguaje de programación Java. El sistema JavaBean amplía el applet[4] de Java para acoplar los componentes de software más sofisticados necesarios para el desarrollo basado en componentes.



Para obtener recursos excelentes de estimación visite la Red de Gestores de Proyectos en **java.sun.com/beans**

[3] En [ORF96] y [YOU98] se presenta un estudio excelente sobre los estándares de «objetos distribuidos».

[4] En este contexto, un applet se puede considerar un componente simple.

> El sistema de componentes JavaBean acompaña un conjunto de herramientas llamadas ***Kit de Desarrollo** Bean* (BDK), que permite a los desarrolladores: (1) analizar el funcionamiento de los Beans (componentes); *(2)* personalizar **su** comportamiento y aspecto; (3) establecer mecanismos de coordinación y comunicación; **(4)** desarrollar Beans personalizados para **su** utilización en una aplicación específica; y (5) probar y evaluar el comportamiento de un Bean.

¿Cuál de estos estándares dominará la industria? Por el momento, no hay una respuesta fácil. Aunque muchos desarrolladores han establecido normas en base a uno de estos estándares, es probable que grandes empresas de software tengan la posibilidad de elegir entre tres estándares, dependiendo de las categorías y plataformas de aplicación que hayan seleccionado.

27.4.2. Ingeniería de componentes

Como se ha señalado anteriormente en este capítulo, el proceso de **ISBC** fomenta la utilización de los componentes de software existentes. Sin embargo, hay ocasiones en que se deben diseñar los componentes. Es decir, los componentes de software nuevos deben desarrollarse e integrarse con los componentes CYD ya existentes y los de desarrollo propio. Dado que estos componentes nuevos van a formar parte de la biblioteca de desarrollo propio de componentes reutilizables, deberían diseñarse para **su** reutilización.

No hay nada de mágico en la creación de componentes de software que se pueden reutilizar. Conceptos de diseño tales como abstracción, ocultamiento, independencia funcional, refinamiento y programación estructurada, junto con los métodos orientados a objetos, pruebas, **SQA** y métodos de verificación de corrección, todos ellos contribuyen a la creación de componentes de software que son reutilizables[5]. En esta sección no se revisarán estos temas, sino que, por el contrario, se tendrán en consideración los temas específicos de la reutilización para prácticas sólidas de ingeniería del software.

27.4.3. Análisis y diseño para la reutilización

Los componentes de diseño y análisis se han tratado detalladamente en las Partes Tercera **y** Cuarta de este libro. Los modelos de datos, funcional y de comportamiento (representados mediante una gama de notaciones distintas) se pueden crear con objeto de describir lo que debe realizar una determinada aplicación. **A** continuación, se utilizan unas especificaciones por escrito para describir estos modelos, **y** obtener como resultado final una descripción completa de los requisitos.

Lo ideal sería analizar el modelo de análisis para determinar aquellos elementos del modelo que indican unos componentes reutilizables ya existentes. El problema consiste en extraer información a partir del modelo de requisitos, de forma que pueda llevar a una «correspondencia de especificaciones». Bellinzoni y sus colaboradores [BEL95] describen un enfoque para los sistemas orientados a objetos:

> **Los** componentes se definen y se almacenan como clases de especificación, diseño e implementación con diferentes niveles de abstracción (cada clase es una descripción ya realizada de un producto procedente de aplicaciones anteriores). El conocimiento de especificación -conocimiento de desarrollo— se almacena en la forma de clases que sugieren la reutilización, y que contienen indicaciones para recuperar componentes reutilizables basándose en **su** descripción y también para componerlos y ajustarlos una vez recuperados.



*Aunque la empresa **no** haga **ingeniería** de dominios, hay que ir realizándola a medida que avanza el trabajo. Cuando **construya** el **modelo** de análisis pregúntese: «**¿Es probable** que este **objeto** a función se pueda **encontrar en otras aplicaciones** de este **tipo?»** Si la respuesto es **afirmativa,** puede que el componente ya existo.*

Las herramientas automatizadas se utilizan para explorar el repositorio en un intento de hacer coincidir el requisito indicado en la especificación actual con los descritos para unos componentes reutilizables ya existentes (clases). Las funciones de caracterización (Sección 27.3.2) y las palabras reservadas se emplean como ayuda para hallar componentes potencialmente reutilizables.

Si la correspondencia de especificaciones produce componentes que se ajustan a las necesidades de la aplicación en cuestión, el diseñador puede extraer estos componentes de una biblioteca de reutilización (repositorio) y utilizarlos para diseñar el nuevo sistema. Si no es posible hallar componentes de diseño, el ingeniero del software entonces debe aplicar métodos de diseño convencionales u OO para crearlos. Es en este momento - cuando el diseñador comienza a crear un nuevo componente — en el que debe de considerarse *el diseño para la reutilización* (DPR).



*Aunque existen **asuntos** especiales a tener en cuenta cuando el **objetivo** es la reutilización, hay que **centrarse** en las principios básicos de un buen diseño. Si se siguen estos principias, las probabilidades de reutilización **aumentarán significativamente.***

Tal como se ha indicado anteriormente, el DPR requiere que el ingeniero del software aplique unos sólidos conceptos y principios de diseño del software (Capítulo 13). Pero las características del dominio de la aplicación también tienen que tenerse en cuenta. Binder [BIN93] sugiere un cierto número de asuntos[6] fun-

[5] Para aprender más sobre estos temas, véase desde el Capítulo 13 hasta el 16, y desde el 20 al 22.

[6] En general las preparaciones indicadas para el diseño destinado a la reutilización deberían de llevarse a cabo como parte de la ingeniería del dominio (Sección 27.3).

damentales que deben considerarse como base del diseño para la reutilización:

Datos estándar. Es preciso investigar el dominio de la aplicación, y es preciso identificar las estructuras de datos estándar globales (por ejemplo, las estructuras de archivos o toda una base de datos completa). Entonces se pueden caracterizar todos los componentes de diseño para uso de estas estructuras de datos estándar.

Protocolos de interfaz estándar. Deberían establecerse tres niveles de protocolo de interfaz: la naturaleza de las interfaces intramodulares, el diseño de interfaces técnicas externas (no humanas) y la interfaz hombre-máquina.

Plantillas de programa. El modelo de estructura (Sección 27.3.3) puede servir como plantilla para el diseño arquitectónico de un nuevo programa.

Una vez que se han establecido los datos estándar, las interfaces y las plantillas de programa, el diseñador posee un marco de referencia en el cual puede crear el diseño. Los componentes nuevos que se ajustan a este marco de referencia tendrán una mayor probabilidad de ser posteriormente reutilizables.

Al igual que el diseño, la construcción de componentes reutilizables se basa en métodos de la ingeniería del software que ya se han descrito en otras partes de este libro. La construcción se puede efectuar empleando lenguajes convencionales de programación, de tercera generación, lenguajes de cuarta generación y generadores de código, técnicas de programación visual o herramientas más avanzadas.

27.5 CLASIFICACIÓN Y RECUPERACIÓN DE COMPONENTES

Considere una gran biblioteca universitaria. Para su utilización tiene disponibles decenas de millares de libros, periódicos y otras fuentes de información. Ahora bien, para acceder a estos recursos es preciso desarrollar algún sistema de clasificación. Para recorrer este gran volumen de información, los bibliotecarios han definido un esquema de clasificación que incluye un código de clasificación de la biblioteca, palabras reservadas, nombres de autor y otras entradas de índice. Todas ellas capacitan al usuario para encontrar el recurso necesario de forma rápida y sencilla.

Cita:
Lo mejor después de conocer algo, es saber dónde se encuentra.
Samuel Johnson

Considere ahora un gran depósito de componentes. Residen en él decenas de millares de componentes de software reutilizables. ¿Cómo puede hallar el ingeniero del software el componente que necesita? Para responder a esta pregunta, surge otra pregunta más. ¿Cómo se describen los componentes de software en términos de no ambiguos y fácilmente clasificables? Se trata de cuestiones difíciles y todavía no se ha desarrollado una respuesta definitiva. En esta sección, se exploran las tendencias actuales que capacitarán a los futuros ingenieros del software para navegar por las bibliotecas reutilizables.

27.5.1. Descripción de componentes reutilizables

Un componente de software reutilizable se puede describir de muchas maneras, pero la descripción ideal abarca lo que Tracz [TRA90] ha llamado el *Modelo 3C* –concepto, contenido y contexto–.

El *concepto* de un componente de software es una «descripción de lo que hace el componente» [WHI95]. La interfaz con el componente se describe completamente y su semántica —representada dentro del contexto de pre y postcondiciones— se identifica también. El concepto debe comunicar la intención del componente.



Poro describir un componente reutilizable, se tendrá que describir su concepto, contenido y contexto.

El *contenido* de un componente describe la forma en que se construye el concepto. En esencia, el contenido es una información que queda oculta a los ojos del usuario habitual y que solamente necesita conocer quien intentará modificar ese componente.

El *contexto* sitúa el componente de software reutilizable en el seno de su dominio de aplicabilidad, esto es, mediante la especificación de características conceptuales, operacionales y de implementación, el contexto capacita al ingeniero del software para hallar el componente adecuado para satisfacer los requisitos de la aplicación.



Uno detallado guía de reutilización de componentes se puede descargar de la dirección
webl .ssg.gunter.af.mil/sep/SEPver40/Main.html#GD

Para que resulte útil en un sentido práctico, el concepto, el contenido y el contexto tienen que traducirse

en un esquema de especificación concreto. Se han escrito varias docenas de artículos y trabajos acerca de los esquemas de clasificación para componentes de software reutilizables (por ejemplo, [WHI95] contiene una extensa bibliografía). Los métodos propuestos se pueden descomponer en tres zonas fundamentales: métodos de las ciencias de la documentación y de biblioteconomía, métodos de inteligencia artificial y sistemas de hipertexto. La gran mayoría de los trabajos realizados hasta el momento sugiere la utilización de métodos propios de biblioteconomía para la clasificación de componentes.

La Figura 27.2 presenta una taxonomía de los métodos de indexación en biblioteconomía. Los *vocabularios controlados de indexación* limitan los términos y sintaxis que se pueden utilizar para clasificar los objetos (componentes). Los *vocabularios de indexación no controlados* no imponen restricciones sobre la naturaleza de la descripción. La gran mayoría de los esquemas de clasificación para los componentes de software se encuentran dentro de las tres categorías siguientes:

Clasificación enumerada. Los componentes se describen mediante la definición de una estructura jerárquica en la cual se definen clases y diferentes niveles de subclases de componentes de software. Los componentes en **sí** se enumeran en el nivel más bajo de cualquier ruta de la jerarquía enumerada. Por ejemplo, una jerarquía enumerada para operaciones con ventanas[7] podría ser

```
operaciones ventana
  visualización
     abrir
      basados en menú
          ventanaAbierta
      basados en sistemas
          ventanaSistema
    cerrar
      a través de puntero
              ....
cambio de tamaño
  a través de órdenes
     establecerTamVentana, redimensionadoEs-
     tandar, contraerVentana
  por arrastre
     arrastrarVentana, estirarVentana
reordenación de planos
              ....
desplazamiento
              ....
cierre
              ....
```

La estructura jerárquica de un esquema de clasificación enumerado hace que sea sencillo comprenderlo y utilizarlo. Sin embargo, antes de poder construir una jerarquía, es preciso llevar a cabo una ingeniería del dominio para que esté disponible una cantidad suficiente de conocimientos acerca de las entradas correctas de esa jerarquía.

[7] Solamente se indica un pequeño subconjunto de todas las operaciones posibles.

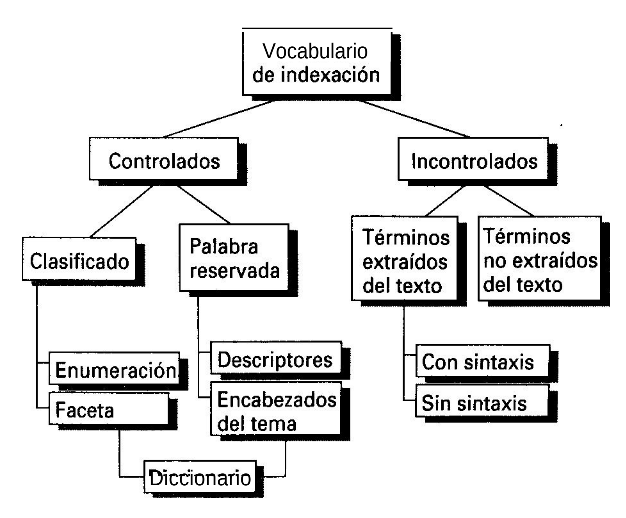
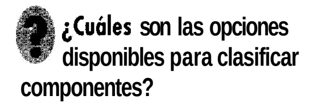


FIGURA 27.2'. Una taxonomía de métodos de indexación [FRA94].

Clasificación por facetas. Se analiza una cierta área del dominio y se identifica un conjunto de características descriptivas básicas. Estas características, que se denominan facetas, reciben entonces diferentes prioridades según su importancia y se relacionan con un componente. La faceta puede describir la función que lleva a cabo el componente, **los** datos que se manipulan, el contexto en que se aplican o cualquier otra característica. El conjunto de facetas que describen los componentes se denomina descriptor de facetas. Generalmente, la descripción por facetas se limita a un máximo de siete u ocho facetas.



Como ilustración sencilla del uso de facetas en la clasificación de componentes, considérese un esquema [LIA93] que hace uso del siguiente descriptor de facetas:

(función, tipo de objeto, tipo de sistema)

Cada una de las facetas del descriptor de facetas adopta uno o más valores que en general serán palabras reservadas descriptivas. Por ejemplo, si función es una faceta de un componente, entonces entre **los** valores típicos que se asignan a esta faceta podríamos contar:

función = (copiar, desde) o bien (copiar, reemplazar, todo)

La utilización de múltiples valores de faceta hace posible que la función primitiva ***copiar*** se refine más exactamente.

Las ***palabras reservadas*** (valores) se asignan al conjunto de facetas de cada componente de una cierta biblioteca de reutilización. Cuando un ingeniero del software desea ocultar la biblioteca en busca de posibles componentes para un diseño, se especifica una lista de valores y se consulta la biblioteca en busca de posibles candidatos. Para incorporar una función de diccionario se pueden emplear herramientas automatizadas. Esto hace posible que la búsqueda no **sólo** abarque las palabras reservadas especificadas por el ingeniero del software, sino también otros sinónimos técnicos de esas mismas palabras reservadas.

Un esquema de clasificación por facetas proporciona al ingeniero del dominio una mayor flexibilidad al especificar descriptores complejos de los componentes [FRA94]. Dado que es posible añadir nuevos valores de facetas con facilidad, el esquema de clasificación por facetas es más fácil de extender y de adaptar que el enfoque de enumeración.

Clasificación de atributos y valores. Para todos los componentes de una cierta zona del dominio se define un conjunto de atributos. A continuación, se asignan valores a estos atributos de forma muy parecida a una clasificación por facetas. De hecho, la clasificación de atributos y valores es parecida a la clasificación por facetas con las excepciones siguientes: (1) no hay límite para el número de atributos que se pueden utilizar; (2) no se asignan prioridades a los atributos; y (3) no se utiliza la función diccionario.

Basándose en un estudio empírico de cada una de las técnicas anteriores de clasificación, Frakes y Pole [FRA94] indican que no existe una técnica que sea claramente «la mejor» y que «ningún método es más que moderadamente adecuado en lo tocante a efectividad de búsqueda.. ..». Parece entonces que es preciso realizar un trabajo más extenso en el desarrollo de unos esquemas de clasificación efectivos para las bibliotecas de reutilización.

27.5.2. El entorno de reutilización

La reutilización de componentes de software debe de estar apoyada por un entorno que abarque los elementos siguientes:

- una base de datos de componentes capaz de almacenar componentes de software, así como la información de clasificación necesaria para recuperarlos.
- un sistema de gestión de bibliotecas que proporciona acceso a la base de datos.
- un sistema de recuperación de componentes de software (por ejemplo, un distribuidor de objetos) que permite a la aplicación cliente recuperar los componentes y servicios del servidor de la biblioteca.
- unas herramientas CASE que prestan su apoyo a la integración de componentes reutilizados en un nuevo diseño o implementación.

Cada una de estas funciones interactúa con una biblioteca de reutilización o bien se encuentra incorporada a esta última.

La biblioteca de reutilización es un elemento de un repositorio CASE más extenso (Capítulo 31) y proporciona las posibilidades adecuadas para el almacenamiento de una amplia gama de elementos reutilizables (por ejemplo, especificación, diseño, casos de prueba, guías para el usuario). La biblioteca abarca una base de datos y las herramientas necesarias para consultar esa base de datos y recuperar de ella componentes. Un esquema de clasificación de componentes (Sección 27.5.1) servirá como base para consultas efectuadas a la biblioteca.

Las consultas suelen caracterizarse mediante el uso del elemento de contexto del Modelo 3C que se describía anteriormente. Si una consulta inicial da lugar a una lista voluminosa de candidatos a componentes, entonces se refina la consulta para limitar esa lista. A continuación, se extraen las informaciones de concepto y contenido (después de haber hallado los candidatos a componentes) para ayudar al desarrollador a que seleccione el componente adecuado.

Una descripción detallada de la estructura de bibliotecas de reutilización y de las herramientas que las gestionan va más allá de los límites de este libro. El lector interesado debería consultar [HOO91] y [LIN95] para más información.

27.6 ECONOMIA DE LA ISBC

La economía de la ingeniería del software basada en componentes tiene un atractivo evidente. En teoría, debería proporcionar a las empresas de software unas ventajas notables en lo tocante a la calidad y a los tiempos de realización. Éstas, a su vez, deberían traducirse en ahorros de costes. ¿Existen datos reales que presten apoyo a nuestra intuición?

Para responder a esta pregunta primero es preciso comprender lo que se puede reutilizar en un contexto de ingeniería del software, y cuáles son los costes asociados a la reutilización. Como consecuencia, será posible desarrollar un análisis de costes y de beneficios para la reutilización.

27.6.1. Impacto en la calidad, productividad y coste

A lo largo de los Últimos años, existen notables evidencias procedentes de casos prácticos en la industria (por ejemplo, [HEN95], {MCM95} y [LIM94]), las cuales indican que es posible obtener notables beneficios de negocios mediante una reutilización agresiva del software. Se mejora tanto la calidad del producto, como la productividad del desarrollador y los costes en general.



ISBC no es un «rompe crismas» económico si los componentes disponibles son /as correctos para el trabajo. Si lo reutilización exige personalizoción, hoy que proceder con precaución.

Calidad. En un entorno ideal, un componente del software que se desarrolle para su reutilización estará verificado en lo tocante a su corrección (véase el Capítulo 26) y no contendrá defectos. En realidad, la verificación formal no se lleva a cabo de forma rutinaria, y los defectos pueden aparecer y aparecen. Sin embargo, con cada reutilización, los defectos se van hallando y

eliminando, y como resultado la calidad del componente mejora. Con el tiempo, el componente llega a estar virtualmente libre de defectos.

En un estudio realizado en Hewlet-Packard, Lim [LIM94] informa que la tasa de defectos para el código reutilizado es de 0,9 defectos por KLDC, mientras que la tasa para el software recién desarrollado es de **4,1** defectos por KLDC. Para que una aplicación esté compuesta por un **68** por 100 de código reutilizado, la tasa de defectos será de **2,0** defectos —una mejora del 51 por **100** para la tasa esperada si la aplicación hubiera sido desarrollada sin reutilización—. Henry y Faller [HEN95] informan de una mejora de calidad del **35** por 100. Aun cuando existen recortes anecdóticos que abarcan un espectro razonablemente amplio en porcentajes de mejora de la calidad, es justo que la reutilización proporcione un beneficio no despreciable en función de la calidad y fiabilidad del software proporcionado.



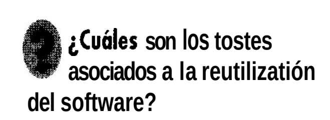
Aunque los datos empíricos varíen, la evidencia industrial indica que la reutilización proporciona un beneficio sustancial en coste.

Productividad. Cuando se aplican componentes reutilizables a lo largo del proceso del software, se invierte menos tiempo creando los planes, modelos, documentos, código y datos necesarios para crear un sistema fiable. Se sigue proporcionando un mismo nivel de funcionalidad al cliente con menos esfuerzo. Consiguientemente, mejora la productividad. Aun cuando los informes acerca de las mejoras de productividad son sumamente difíciles de interpretar[8], parece que una reutilización de entre el **30** y el **50** por 100 puede dar lugar a mejoras de productividad que se encuentren entre el intervalo que media entre el **25** y el 40 por 100.

Coste. El ahorro neto de costes para la reutilización se estima proyectando el coste del proyecto si se hubiera desarrollado éste desde cero, C, y restando después la suma de los costes asociados para la reutilización, C_r, y el coste real del software cuando este finalmente se implanta, C_i.

C se puede determinar aplicando una o más de las técnicas de estimación descritas en el Capítulo 5. Los costes asociados a la reutilización, C_r, incluyen [CHR95]:

- Modelado y análisis del dominio;
- Desarrollo de la arquitectura del dominio;



- Incremento de la documentación para facilitar la reutilización;
- Mantenimiento y mejora de componentes de la reutilización;
- Licencias para componentes adquiridos externamente;
- Creación o adquisición y funcionamiento de un depósito para la reutilización;
- Formación del personal en el diseño y construcción para la reutilización.

Aun cuando los costes asociados al análisis del dominio (Sección **27.4)** y el funcionamiento de un repositorio para la reutilización pueden resultar notables, muchos de los costes restantes indicados anteriormente se enfrentan con temas que forman parte de las buenas prácticas de la ingeniería del software, tanto si se considera prioritaria la reutilización como si no.

27.6.2. Análisis de coste empleando puntos de estructura

En la Sección **27.3.3** se definía un punto de estructura como una trama arquitectónica que aparece repetidamente en el seno de un determinado dominio de aplicaciones. El diseñador del software (o el ingeniero del sistema) puede desarrollar una arquitectura para una nueva aplicación, sistema o producto mediante la definición de una arquitectura del dominio y poblándola entonces con puntos de estructura. Estos puntos de estructura son, o bien componentes reutilizables, o bien paquetes de componentes reutilizables.

Aun cuando los puntos de estructura sean reutilizables, su adaptación, integración y costes de mantenimiento no serán despreciables. Antes de seguir adelante con la reutilización, el gestor del proyecto debe entender los costes asociados al uso de puntos estructura.

Dado que todos los puntos de estructura (y todos los componentes reutilizables en general) poseen una historia pasada, es posible recoger datos de costes para cada uno de ellos. En una situación ideal, los costes de adaptación, integración y mantenimiento asociados a los componentes de una biblioteca de reutilización se mantienen para cada caso de utilización. Entonces es posible analizar estos datos para desarrollar una estimación de costes para el próximo caso en que se utilicen.

Como ejemplo, considérese una nueva aplicación, X, que requiere un 60 por 100 de código nuevo, y la reutilización de tres puntos de estructura, PE,, PE, y PE,. Cada'uno de estos componentes reutilizables ha sido

[8] Existen muchas circunstancias atenuantes (por ejemplo, área de aplicación, complejidad del problema, estructura y tamaño del equipo, duración del proyecto, tecnología aplicada) que puede tener su impacto sobre la productividad del equipo de un proyecto.

utilizado en un cierto número de aplicaciones adicionales, y los costes medios de adaptación, integración y mantenimiento están disponibles.

¿Existe un cálculo rápido que permita estimar el beneficio de la reutilización de componentes en términos de coste?

Para estimar el esfuerzo requerido para proporcionar la aplicación, X, es preciso efectuar el siguiente cálculo:

$$\text{esfuerzo global} = E_{nuevo} + E_{igual} + E_{adaptación} + E_{integración}$$

donde

E_{nuevo} = es el esfuerzo que el ingeniero requiere para construir los nuevos componentes de software (que se determinará empleando las técnicas descritas en el Capítulo 5);

E_{igual} = es el esfuerzo requerido para cualificar PE,, PE_2 y PE_3;

$E_{adaptación}$ = es el esfuerzo requerido para adaptar PE,, PE, y PE_3;

$E_{integración}$ = es el esfuerzo requerido para integrar PE,, PE, y PE_3;

El esfuerzo necesario para cualificar, adaptar e integrar PE,, PE, y PE, se determina calculando la media de datos históricos recogidos para la cualificación, adaptación e integración de los componentes reutilizables en otras aplicaciones.

27.6.3. Métricas de reutilización

Se ha desarrollado toda una gama de métricas de software en un intento de medir los beneficios de la reutilización en un sistema basado en computadoras. Los beneficios asociados a la reutilización dentro de un sistema S se pueden expresar como el siguiente cociente

$$R_b(S) = [C_{\text{sin reutilización}} - C_{\text{con reutilización}}] / C_{\text{sin reutilización}} \quad (27.1)$$

donde

$C_{\text{sin reutilización}}$ es el coste de desarrollar S sin reutilización, y

$C_{\text{con reutilización}}$ es el coste de desarrollar S con reutilización

De lo que se deduce que $R_b(S)$ se puede expresar como un valor no dimensional que se encuentra en el intervalo

$$0 \le R_b(S) \le 1 \quad (27.2)$$

Devanbu y sus colaboradores [DEV95] sugieren que (1) R, se verá afectado por el diseño del sistema; (2) dado que R, se ve afectado por el diseño, es importante hacer que R, forme parte de la estimación de alternativas de diseño; y **(3)** los beneficios asociados a la reutilización estarán íntimamente relacionados con los beneficios en términos de costes de cada uno de los componentes reutilizables individuales.

Se define una medida general de reutilización en los sistemas orientados a objetos, denominada *aprovechamiento **de** reutilización* [BAN94] en la siguiente forma:

$$R_{\text{aprovechado}} = OBJ_{\text{reutilizados}} / OBJ_{\text{construidos}} \quad (27.3)$$

donde

$OBJ_{\text{reutilizados}}$ es el número de objetos reutilizados en el sistema;

$OBJ_{\text{construidos}}$ es el número de objetos construidos para el sistema.

De lo que se deduce que a medida que aumenta $R_{\text{aprovechado}}$, también aumenta R,.

RESUMEN

La ingeniería del software basada en componentes proporciona unos beneficios inherentes en lo tocante a calidad del software, productividad del desarrollador **y** coste general del sistema. Sin embargo, es preciso vencer muchas dificultades antes de que el modelo de proceso ISBC se utilice ampliamente en la industria.

Además de los componentes del software, un ingeniero del software puede adquirir toda una gama de elementos reutilizables. Entre estos se cuentan las representaciones técnicas del software (por ejemplo, especificación, modelos de arquitectura, diseños y códigos), documentos, datos de prueba e incluso tareas relacionadas con los procesos (por ejemplo, técnicas de inspección).

El proceso ISBC acompaña a dos subprocesos concurrentes —ingeniería del dominio **y** desarrollo basado en componentes—. El objetivo de la ingeniería del dominio es identificar, construir, catalogar y diseminar un conjunto de componentes de software en un determinado dominio de aplicación. **A** continuación, el desarrollo basado en componentes cualifica, adapta e integra estos componentes para su reutilización en un sistema nuevo. Además, el desarrollo basado en componentes diseña componentes nuevos que se basan en los requisitos personalizados de un sistema nuevo.

Las técnicas de análisis y diseño de componentes reutilizables se basan en los mismos principios y conceptos que forman parte de las buenas prácticas de ingeniería del software. Los componentes reutilizables deberían de diseñarse en el seno de un entorno que establezca unas estructuras de datos estándar, unos protocolos de interfaz y unas arquitecturas de programa para todos los dominios de aplicación.

La ingeniería del software basada en componentes hace uso de un modelo de intercambio de datos; herramientas, almacenamiento estructurado y un modelo objeto subyacente para construir aplicaciones. El modelo de objetos suele ajustarse a uno o más estándares de componentes (por ejemplo, OMG/CORBA) que definen la forma en que una aplicación puede acceder a los

objetos reutilizables. Los esquemas de clasificación capacitan al desarrollador para hallar y recuperar componentes reutilizables y se ajustan a un modelo que identifica conceptos, contenidos y contextos. La clasificación enumerada, la clasificación de facetas, y la clasificación de valores de atributos son representativas de muchos esquemas de clasificación de componentes.

La economía de reutilización del software se abarca con una Única pregunta: ¿Es rentable construir menos y reutilizar más? En general, la respuesta es «sí», pero un planificador de proyectos de software debe considerar los costes no triviales asociados a la adaptación e integración de los componentes reutilizables.

REFERENCIAS

[ADL95] Adler, R.M., «Engineering Standards for Component Software», *Computer*, vol. 28, n.° 3, Marzo de 1995, pp. 68-77.

[BAS94] Basili, V.R., L.C. Briand y W.M. Thomas, «Domain Analysis for the Reuse of Software Development Experiences», *Proc. Of the 19th Annual Software Engineering Workshop*, NASA/GSFC, Greenbelt, MD, Diciembre de 1994.

[BEL95] Bellinzona, R., M.G. Gugini y B. Pernici, «Reusing Specifications in OO Applications», *IEEE Software*, Marzo de 1995, pp. 65-75.

[BIN93] Binder, R., «Design for Reuse is for Real», *American Programmer*, vol. 6, n.° **8,** Agosto de 1993, pp. 33-37.

[BRO96] Brown, A.W., y K.C. Wallnau, «Engineering of Component Based Systems», *Component-Based Software Engineering*, IEEE Computer Society Press, 1996, pp. 7-15.

[CHR95] Christensen, S.R., «Software Reuse Iniciatives at Lockheed», *Crosstalk*, vol. 8, n.° **5,** Mayo de 1995, pp. 26-31.

[CLE95] Clemens, P.C., «From Subroutines to Subsystems: Component-Based Software Development», *American Programmer*, vol. 8, n.° 11, Noviembre de 1995.

[DEV95] Devanbu, P. et al., «Analytical and Empirical Evaluation of Software Reuse Metrics», *Technical Report*, Computer Science Department, University of Maryland, Agosto de 1995.

[FRA94] Frakes, W.B., y T.P. Pole, «An Empirical Study of Representation Methods for Reusable Software Components», *IEEE Trans. Software Engineering*, vol. 20, n.° 8, Agosto de 1994, pp. 617-630.

[HAR98] Harmon, P., «Navigating the Distributed Components Landscape», *Cutter IT Journal*, vol. 11, **n.° 2,** Diciembre de 1998, pp. 4-11.

[HEN95] Henry, E., y B. Faller, «Large Scale Industrial Reuse to Reduce Cost and Cycle Time», *IEEE Software*, Septiembre de 1995, pp. 47-53.

[HOO91] Hooper, J.W., y R.O. Chester, *Software Reuse: Guidelines and Methods*, Plenum Press, 1991.

[HUT88] Hutchinson, J.W., y P.G. Hindley, «A Preliminary Study of Large Scale Software Reuse», *Software Engineering Journal*, vol. 3, n.° **5,** 1998, pp. 208-212.

[LIM94] Lim, W.C., «Effects of Reuse on Quality, Productivity, and Economics», *IEEE Software*, Septiembre de 1994, pp. 23-30.

[LIA93] Liao, H., y F. Wang, «Software Reuse Based on a Large Object-Oriented Library», *ACM Software Engineering Notes*, vol. 18, n.° 1, Enero de 1993, pp. 74-80.

[LIN95] Linthicum, D.S., «Component Development (a special feature)», *Application Development Trends*, Junio de 1995, pp. 57-78.

[MCM95] McMahon, P.E., «Pattern-Based Architecture: Bridging Software Reuse and Cost Management», *Crosstalk*, vol. 8, n.° 3, Marzo de 1995, **pp.** 10-16.

[ORF96] Orfali, R., D. Harkey y J. Edwards, *The Essential Distributed Objects Survival Guide*, Wiley, 1996.

[PRI87] Prieto-Díaz, R., «Domain Analysis for Reusability», *Proc. COMPSAC* '87, Tokyo, Octubre de 1987, pp. 23-29.

[PRI93] Prieto-Díaz, R., «Issues and Experiences in Software Reuse», *American Programmer*, vol. 6, n.° 8, Agosto de 1993, pp. 10-18.

[POL94] Pollak, W., y M. Rissman, «Structural Models and Pattemed Architectures», *Computer*, vol. 27, n.° 8, Agosto de 1994, pp. 67-68.

[STA94] Staringer, W., «Constructing Applications from Reusable Components», *IEEE Software*, Septiembre de 1994, pp. 61-68.

[TRA90] Tracz, W., «Where Does Reuse Start?», *Proc. Realities of Reuse Workshop*, Syracuse University CASE Center, Enero de 1990.

[TRA95] Tracz, W., «Third Intemational Conference on Software Reuse-Summary», *ACM Software Engineering Notes*, vol. 20, n.° 2, Abril de 1995, pp. 21-22.

[WHI95] Whittle, B., «Models and Languages for Component Description and Reuse», *ACM Software Engineering Notes*, vol. 20, n.° 2, Abril de 1995, pp. 76-89.

[YOU98] Yourdon, E. (ed.), «Distributed Objects», *Cutter IT Journal*, vol. 11, **n.º** 12, Diciembre de 1998.

PROBLEMAS Y PUNTOS A CONSIDERAR

27.1. Uno de los obstáculos principales de la reutilización consiste en hacer que **los** desarrolladores de software consideren la reutilización de componentes ya existentes, en lugar de volver a inventar otros nuevos. (¡Después de todo, es divertido construir cosas!) Sugiera tres o cuatro formas distintas en que una organización de software pueda proporcionar incentivos para que los ingenieros de software utilicen la reutilización. ¿Qué tecnologías deberían de estar implantadas para apoyar ese esfuerzo de reutilización?

27.2. Aunque los componentes de software son los «elementos» reutilizables más obvios, se pueden reutilizar muchas otras entidades producidas como parte de la ingeniería del software. Tenga en consideración los planes de proyecto **y** las estimaciones de costes. ¿Cómo se pueden reutilizar y cuál es el beneficio de hacerlo?

27.3. Lleve a cabo una pequeña investigación acerca de la ingeniería de dominios, y detalle algo más el modelo de procesos esbozado en la Figura 27.1. Identifique las **tareas** necesarias para el análisis de dominios y para el desarrollo de arquitecturas de software.

27.4. ¿En qué sentido son iguales las funciones de caracterización para los dominios de aplicaciones y los esquemas de clasificación de componentes? ¿En qué sentido son distintas?

27.5. Desarrolle un conjunto de características del dominio para sistemas de información que sean relevantes para el procesamiento de datos de alumnos de una Universidad.

27.6. Desarrolle un conjunto de características del dominio que sean relevantes para el software de publicación y autoedición.

27.7. Desarrolle un modelo estructural sencillo para un dominio de aplicación que le asigne su instructor, o bien uno con el cual esté familiarizado.

27.8. ¿Qué es un punto de estructura?

27.9. Obtenga información de los estándares más recientes de CORBA, COM o de JavaBeans y prepare un trabajo de 3 a 5 páginas que trate los puntos más destacables. Obtenga información de una herramienta de distribución de solicitudes de objetos e ilustre en que esa herramienta se ajusta al estándar.

27.10. Desarrolle una clasificación enumerada para un dominio de aplicación que le asigne su instructor o para uno con el cual esté familiarizado.

27.11. Desarrolle un esquema de clasificación por facetas para un dominio de aplicación que le asigne **su** instructor o bien para uno con el cual esté familiarizado.

27.12. Investigue en la literatura para conseguir datos recientes de calidad y productividad que apoyen la utilización. Presente esos datos al resto de la clase.

27.13. Se estima que un sistema orientado a objetos requiere 320 objetos para su finalización. Además, se estima que es posible adquirir 190 objetos procedentes de un depósito ya existente. ¿Cuál es el aprovechamiento de reutilización? Suponga que los objetos nuevos cuestan 260.000 pts., y que se necesitan 156.000 pts. para adaptar un objeto y 104.000 pts. para integrarlo. ¿Cuál es el coste estimado del sistema? ¿Cuál es el valor de Rb?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Durante los últimos años se han publicado libros sobre el desarrollo basado en componentes y la reutilización de componentes. Allen, Frost y Yourdon *(Component-Based Developmentfor Enterprise Systems: Applying the Select Perspective,* Cambridge University Press, 1998) abarca todo el proceso de ISBC mediante el uso de UML (Capítulos 21 y 22) basándose en el enfoque del modelado. Los libros de Lim *(Managing Software Reuse: A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components,* Prentice-Hall, 1998), Coulange *(Software Reuse,* Spriger Verlag, 1998), Reifer *(Practica1Software Reuse,* Wiley, 1997), Jacobson, Griss y Jonsson *(Software Reuse: Architecture Process and Organizationfor Business Success*, Addison-Wesley, 1997) afronta muchos temas de ISBC. Fowler *(Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1996) considera la aplicación de patrones arquitectónicos dentro del proceso de ISBC y proporciona muchos ejemplos Útiles. Tracz *(Confessions of a Used Program Salesman: Institutionalizing Software Reuse*, Addison-Wesley, 1995) presenta un estudio algunas veces desenfadado y muy útil de los temas asociados con la creación de una cultura de reutilización.

Leach *(Software Reuse: Methods, Models and Costs,* McGraw-Hill, 1997) proporciona un análisis detallado de los estudios de costes asociados con la ISBC y con la reutilización. Poulin *(Measuring Software Reuse: Principles, Practices, and Economic Models,* Addison-Wesley, 1996) sugiere un número de métodos cuantitativos para valorarlos beneficios de la reutilización del software.

Durante los Últimos años se han publicado docenas de libros que describen los mismos estándares basados en componentes de la industria, pero también tienen en consideración muchos tópicos importantes de la ISBC. A continuación se muestra un muestreo del estudio de los tres estándares:

CORBA

Doss, G.M., *CORBA Networking Java*, Wordware Publishing, 1999.

Hoque, R., *CORBAfor Real Prograrnmers,* Academic Press/Morgan Kaufmann, 1999.

Siegel, **J.**, *CORBA **3** Fundamentals and Programming,* Wiley, 1999.

Slama, D., J. Garbis y P. Russell, *Enterprise CORBA,* Prentice-Hall, 1999.

COM

Box, D.K., T. Ewald y C. Sells, *Effective Ways to Improve Your COM and MTS-Based Applications,* Addison-Wesley, 1999.

Kirtland, M., *Designing Component-Based Applications,* Microsoft Press, 1999.

Muchas compañías aplican una combinación de estándares de componentes. Los libros de Geraghty et al. *(COM-CORBA*

Interoperubility, Prentice-Hall, 1999), Pritchard *(COM and CORBA Side by Side: Architectures, Strutegies, und Implementations*, Addsion-Wesley, 1999) y Rosen et al. *(Integrating CORBA und COM Applicutions*, Wiley, 1999) tienen en consideración **los** temas asociados con la utilización de CORBA y COM como base para el desarrollo basado en componentes.

JavaBeans

Asbury, S., y S.R. Weiner, *Developing Javu Enterprise Applications*, Wiley, 1999.

Valesky, **T.C.,** *Enterprise JavaBeans: Developing Component-Based Distributed Applicutions*, Addison-Wesley, 1999.

Vogel, A., y M. Rangarao, *Programming with Enterprise JavaBeans, JTS und OTS*, Wiley, 1999.

En Intemet está disponible una gran variedad de fuentes de información sobre la ingeniería del software basada en componentes. Una lista actualizada de referencias en la red que son relevantes para la ISBC se puede encontrar en la dirección **http://www.pressman5.com**

CAPÍTULO

28 INGENIERÍA DEL SOFTWARE DEL COMERCIO ELECTRÓNICO CLIENTE / SERVIDOR

A finales de siglo, el desarrollo de una nueva generación de máquinas herramientas capaces de soportar fuertes tolerancias dieron poder a los ingenieros que diseñaban un proceso nuevo de fabricación llamado producción en masa. Antes de la llegada de esta tecnología avanzada de máquinas herramientas, no se podían soportar fuertes tolerancias. Pero con esta tecnología se podían construir piezas intercambiables y fácilmente ensamblables —la piedra angular de la producción en masa—.

Cuando se va a desarrollar un sistema basado en computadora, un ingeniero de software se ve restringido por las limitaciones de las tecnologías existentes y potenciado cuando las tecnologías nuevas proporcionan capacidades que no estaban disponibles para las generaciones anteriores de ingenieros. La evolución de las arquitecturas distribuidas de computadora ha capacitado a los ingenieros de sistemas y del software para desarrollar nuevos enfoques sobre cómo se estructura el trabajo y cómo se procesa la información dentro de una empresa.

Las nuevas estructuras de las organizaciones y los nuevos enfoques de proceso de información (por ejemplo: tecnologías intranet e Intemet, sistemas de apoyo a las decisiones, software de grupo, e imágenes) representan una salida radical de las primeras tecnologías basadas en minicomputadoras o en mainframes. Las nuevas arquitecturas de computadora han proporcionado la tecnología que ha hecho posible que las empresas vuelvan a diseñar sus procesos de negocio (Capítulo 30).

En este capítulo, examinaremos una arquitectura dominante para el proceso de información —los sistemas *cliente/servidor* (C/S) dentro del contexto de los sistemas de comercio electrónico—. Los sistemas cliente/servidor han evolucionado junto con los avances de la informática personal, en la ingeniería del software basada en componentes, con las nuevas tecnologías de almacenamiento, comunicación mejorada a través de redes, y tecnología de bases de datos mejoradas. El objetivo de este capítulo[1] es presentar una visión global y breve de los sistemas cliente/servidor con un énfasis especial en los temas de la ingeniería del software que deben de afrontarse cuando se analizan, diseñan, prueban y se da soporte a dichos sistemas C/S.

VISTAZO RÁPIDO

¿Qué es? Las arquitecturas cliente/servidor (C/S) dominan el horizonte de los sistemas basados en computadora. Todo existe: desde redes de cajeros automáticos hasta Internet. y esto es debido a que el software que reside en una computadora - el cliente— solicita servicios y/o datos de otra computadora - el servidor—. La ingeniería del software cliente/servidor combina principios convencionales, conceptos y métodos tratados anteriormente en este libro, con elementos de la ingeniería del software basada en componentes y orientada a objetos para crear sistemas C/S.

¿Quién lo hace? Los ingenieros de software llevan a cabo el análisis, diseño, implementación y prueba de estos sistemas.

¿Por qué es importante? El impacto de los sistemas C/S en los negocios, el comercio. el gobierno y la ciencia es dominante. Puesto que los avances tecnológicos (por ejemplo, desarrollo basado en componentes, agentes de solicitud de objetos, Java) cambian la forma de construir los sistemas C/S, en su construcción se debe de aplicar un proceso de ingeniería del software sólido.

¿Cuáles son los pasos? Los pasos que se siguen en la ingeniería de los sistemas C/S son similares a los que se aplican durante la ingeniería del software basada en componentes y OO. El modelo de proceso es evolutivo, comenzando por la obtención de los requisitos. La funcionalidad se asigna a los subsistemas de componentes que se van a asignar a después o bien al cliente o al servidor de la arquitectura C/S. El diseño se centra en la integración de los componentes existentes y en la creación de componentes nuevos. La implementación y las pruebas luchan por ejercitar la funcionalidad del cliente y del servidor dentro del contexto de los estándares de integración de componentes y de la arquitectura C/S.

¿Cuál es el producto obtenido? Un sistema C/S de alta calidad es el producto de la ingeniería del software C/S. También se producen otros productos de trabajo de software (tratados anteriormente en este mismo libro).

¿Cómo puedo estar seguro de que lo he hecho correctamente? Utilizando las mismas prácticas SQA que se aplican en todos los procesos de ingeniería del software —las revisiones técnicas formales evalúan los modelos de análisis y diseño—; las revisiones especializadas consideran los temas asociados a la integración de componentes y al software intermedio (*middleware*), y las pruebas se aplican para desvelar errores al nivel de componentes, subsistema, cliente y servidor.

[1] Parte de este capítulo se ha adaptado a partir del material de cursos desarrollado por John Porter para el Client/Server Curriculum ofrecido en la Facultad de Ingeniería BEI de la Universidad de Fairfield. Se ha obtenido permiso para su utilización.

28.1 INTRODUCCIÓN

Los últimos diez años han sido testigos de avances masivos en las áreas de computación. La primera es que el hardware se ha ido abaratando cada vez más, y a su vez se ha ido haciendo más potente: las computadoras de sobremesa hoy en día tienen la potencia que poseían los mainframes hace algunos años. La segunda área es la de las comunicaciones; avances tales como los sistemas de comunicación vía satélite y los sistemas de teléfonos digitales significa que ahora es posible conectar económicamente y eficientemente con otros sistemas informáticos separados físicamente. Esto ha llevado al concepto de un sistema de computación distribuido. Dicho sistema consiste en un número de computadoras que están conectadas y que llevan a cabo diferentes funciones. Existen muchas razones por las que tales sistemas se han hecho populares:

Rendimiento. El rendimiento de muchos tipos de sistemas distribuidos se puede incrementar añadiendo simplemente más computadoras. Normalmente esta es una opción más sencilla y más barata que mejorar un procesador en un mainframe. Los sistemas típicos en donde se puede lograr este incremento en el rendimiento son aquellos en donde las computadoras distribuidas llevan a cabo mucho proceso, y en donde la relación trabajo de comunicaciones y proceso es bajo.

Compartición de recursos. Un sistema distribuido permite a sus usuarios acceder a grandes cantidades de datos que contienen las computadoras que componen el sistema. En lugar de tener que reproducir los datos en todas las computadoras se pueden distribuir por un pequeño número de computadoras. Un sistema distribuido también proporciona acceso a servicios especializados que quizás no se requieran muy frecuentemente, y que se puedan centralizar en una computadora del sistema.

Cita:
El modelo de computación C/S representa un ejemplo específico de proceso cooperativo distribuido, en donde la relación entre clientes y servidores es la relación de los componentes tanto del hardware como del software.
Alex Berson

Tolerancia a fallos. Un sistema distribuido se puede diseñar de forma que tolere los fallos tanto del hardware como del software. Por ejemplo, se pueden utilizar varias computadoras que lleven a cabo la misma tarea en un sistema distribuido. Si una de las computadoras funcionaba mal, entonces una de sus hermanas puede hacerse cargo de su trabajo. Una base de datos de una computadora se puede reproducir en otras computadoras de forma que si la computadora original tiene un mal funcionamiento, los usuarios que solicitan la base de datos son capaces de acceder a las bases de datos reproducidas.

28.2 SISTEMAS DISTRIBUIDOS

28.2.1. Clientes y servidores

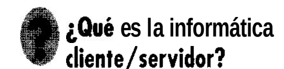
El propósito de esta sección es introducir tanto la idea de cliente como la de servidor. Estos son los bloques básicos de construcción de un sistema distribuido y, de esta manera, cuando se describa el diseño y el desarrollo de dichos sistemas, será necesario tener conocimiento de sus funciones y de su capacidad.

Un servidor es una computadora que lleva a cabo un servicio que normalmente requiere mucha potencia de procesamiento. Un cliente es una computadora que solicita los servicios que proporciona uno o más servidores y que también lleva a cabo algún tipo de procesamiento por **sí** mismo. Esta forma de organización de computadoras es totalmente diferente a los dos modelos que dominaron los años ochenta y principios de los noventa.

El primer modelo se conoce como procesamiento central (*host*). En este modelo de organización todo el procesamiento que se necesitaba para una organización se llevaba a cabo por una computadora grande —normalmente una mainframe— mientras que los usuarios emplean sencillas terminales informáticas o PCs de muy poca potencia para comunicarse con el central. Los dos problemas más serios son la dificultad de mejorar y de copiar con interfaces IGU modernas. **A** medida que las aplicaciones van siendo más grandes, la carga de una mainframe común llega al punto en que también necesita mejorar y normalmente con hardware de procesamiento nuevo, memoria o almacenamiento de archivos. Mejorar dichas computadoras es más fácil que antes; sin embargo, puede resultar un proceso moderadamente difícil y caro -es ciertamente más caro y más difícil que añadir un servidor nuevo basado en PC a un conjunto de computadoras configuradas como clientes y servidores—. El segundo problema es hacerse con interfaces IGU modernas. Para ordenar a una computadora que visualice incluso una pantalla relativamente primitiva relacionada, digamos, con unos cuantos botones y una barra de desplazamiento de la misma, conlleva tanto tráfico en las líneas de comunicación que un sistema podría colapsarse fácilmente con los datos que se utilizan para configurar y manterer una serie de interfaces basadas en IGUs.

El segundo modelo de computación es en donde hay un grupo de computadoras actuando como servidores,

pero poseen poco procesamiento que llevar a cabo. Normalmente estos terminales poco inteligentes actuarían como servidores de archivos o servidores de impresión para un número de PCs potentes o minicomputadoras que llevarían a cabo el procesamiento y estarían conectados a una red de área local. Las computadoras cliente solicitarían servicios a gran escala, como es obtener un archivo, llevando a cabo entonces el procesamiento de dicho archivo. De nuevo, esto conduce a problemas con el tráfico en donde, por ejemplo, la transmisión de archivos grandes a un número de clientes que requieren simultáneamente estos archivos hace que el tiempo de respuesta de la red vaya tan lento como una tortuga.



La computación cliente/servidor es un intento de equilibrar el proceso de una red hasta que se comparta la potencia de procesamiento entre computadoras que llevan a cabo servicios especializados tales como acceder a bases de datos (servidores), y aquellos que llevan a cabo tareas tales como la visualización IGU que es más adecuado para el punto final dentro de la red. Por ejemplo, permite que las computadoras se ajusten a tareas especializadas tales como el procesamiento de bases de datos en donde se utilizan hardware y software de propósito especial para proporcionar un procesamiento rápido de la base de datos comparado con el hardware que se encuentra en las mainframes que tienen que enfrentarse con una gran gama de aplicaciones.

28.2.2. Categorías de servidores

Ya se ha desarrollado una gran variedad de servidores. La siguiente lista ampliada se ha extraído de [ORF99]:

Servidores de archivos. Un servidor de archivos proporciona archivos para clientes. Estos servidores se utilizan todavía en algunas aplicaciones donde los clientes requieren un procesamiento complicado fuera del rango normal de procesamiento que se puede encontrar en bases de datos comerciales. Por ejemplo, una aplicación que requiera el almacenamiento y acceso a dibujos técnicos, digamos que para una empresa de fabricación, utilizaría un servidor de archivos para almacenar y proporcionar los dibujos a los clientes. Tales clientes, por ejemplo, serían utilizados por ingenieros quienes llevarían a cabo operaciones con dibujos, operaciones que serían demasiado caras de soportar, utilizando una computadora central potente. Si los archivos solicitados no fueran demasiado grandes y no estuvieran compartidos por un número tan grande de usuarios, un servidor de archivos sería una forma excelente de almacenar y procesar archivos.

Servidores de bases de datos. Los servidores de bases de datos son computadoras que almacenan grandes colecciones de datos estructurados. Por ejemplo, un banco utilizaría un servidor de bases de datos para almacenar registros de clientes que contienen datos del nombre de cuenta, nombre del titular de la cuenta, saldo actual de la cuenta y límite de descubierto de la cuenta. Una de las características de las bases de datos que invalidan la utilización de los servidores de archivos **es** que los archivos que se crean son enormes y ralentizan el tráfico si se transfirieran en bloque al cliente. Afortunadamente, para la gran mayoría de aplicaciones no se requiere dicha transferencia; por ejemplo en una aplicación bancaria las consultas típicas que se realizarían en una base de datos bancaria serían las siguientes:

- Encontrar las cuentas de los clientes que tienen un descubierto mayor de 2.000 pts.
- Encontrar el saldo de todas las cuentas del titular John Smith.
- Encontrar todas las órdenes regulares del cliente Jean Smith.
- Encontrar el total de las transacciones que se realizaron ayer en la sucursal de Manchester Picadilly.

Actualmente una base de datos bancaria típica tendrá millones de registros, sin embargo, las consultas anteriores conllevarían transferir datos a un cliente que sería solamente una fracción muy pequeña del tamaño.

En un entorno de bases de datos cliente/servidor los clientes envían las consultas a la base de datos, normalmente utilizando alguna IGU. Estas consultas se envían al servidor en un lenguaje llamado SQL (Lenguaje de Consultas Estructurado). El servidor de bases de datos lee el código SQL, lo interpreta y, a continuación, lo visualiza en algún objeto HCI tal como una caja de texto. El punto clave aquí es que el servidor de bases de datos lleva a cabo todo el procesamiento, donde el cliente lleva a cabo los procesos de extraer una consulta de algún objeto de entrada, tal como un campo de texto, enviar la consulta y visualizar la respuesta del servidor de la base de datos en algún objeto de salida, tal como un cuadro de desplazamiento.

Servidores de software de grupo. Software de grupo es el término que se utiliza para describir el software que organiza el trabajo de un grupo de trabajadores. Un sistema de software de grupo normalmente ofrece las siguientes funciones:

- Gestionar la agenda de los individuos de un equipo de trabajo.
- Gestionar las reuniones para un equipo, por ejemplo, asegurar que todos los miembros de un equipo que tienen que asistir a una reunión estén libres cuando se vaya a celebrar.
- Gestionar el flujo de trabajo, donde las tareas se distribuyen a los miembros del equipo y el sistema de software de grupo proporciona información sobre la finalización de la tarea y envía un recordatorio al personal que lleva a cabo las tareas.
- Gestionar el correo electrónico, por ejemplo, organizar el envío de un correo específico a los miembros de un equipo una vez terminada una tarea específica.

Un servidor de software de grupo guarda los datos que dan soporte a estas tareas, por ejemplo, almacena las listas de correos electrónicos y permite que los usuarios del sistema de software de grupo se comuniquen con él, notificándoles, por ejemplo, que se terminado una tarea o proporcionándoles una fecha de reunión en la que ciertos empleados puedan acudir.



Servidores Web. Los documentos Web se almacenan como páginas en una computadora conocida como servidor Web. Cuando se utiliza un navegador (*browser*) para ver las páginas Web normalmente pincha sobre a enlace en un documento Web existente. Esto dará como resultado un mensaje que se enviará al servidor Web que contiene la página. Este servidor responderá entonces enviando una página a su computadora, donde el navegador pueda visualizarlo. De esta manera los servidores Web actúan como una forma de servidor de archivos, administrando archivos relativamente pequeños a usuarios, quienes entonces utilizan un navegador para examinar estas páginas. Para comunicarse con un navegador Web, un cliente que utiliza un navegador está haciendo uso a su vez de un protocolo conocido como HTTP.

Servidores de correo. Un servidor de correo gestiona el envío y recepción de correo de un grupo de usuarios. Para este servicio normalmente se utiliza un PC de rango medio. Existen varios protocolos para el correo electrónico. Un servidor de correo estará especializado en utilizar solo uno de ellos.

Servidores de objetos. Uno de los desarrollos más excitantes en la informática distribuida durante los últimos cinco años ha sido el avance realizado, tanto por parte de los desarrolladores, como por parte de los investigadores, para proporcionar objetos distribuidos. Estos son objetos que se pueden almacenar en una computadora, normalmente un servidor, con clientes capaces de activar la funcionalidad del objeto enviando mensajes al objeto, los cuales se corresponden con métodos definidos por la clase de objeto. Esta tecnología liberará finalmente a los programadores de la programación de bajo nivel basada en protocolos requerida para acceder a otras computadoras de una red. En efecto, esto permite que el programador trate a los objetos a 'distancia como si estuvieran en su computadora local. Un servidor que contiene objetos que pueden accederse a distancia se conoce como servidor de objetos.

Servidores de impresión. Los servidores de impresión dan servicio a las solicitudes de un cliente remoto. Estos servidores tienden a basarse en PCs bastante baratos, y llevan a cabo las funciones limitadas de poner en cola de espera las peticiones de impresión, ordenar a la impresora que lleve a cabo el proceso de impresión e informar a las computadoras cliente que ya ha finalizado una petición de impresión en particular.

Servidores de aplicaciones. Un servidor de aplicaciones se dedica a una aplicación única. Tales servidores suelen escribirse utilizando un lenguaje tal como Java o C++. Un ejemplo típico del servidor que se utiliza en el dibujo de un fabricante de aviones que gestionaba las versiones diferentes de dibujos producidos por el personal técnico iría dirigido a algún proceso de fabricación.

28.2.3. Software intermedio (middleware)

Hasta el momento probablemente ya tenga la impresión de que la comunicación entre cliente y servidor es directa. Desgraciadamente, esto no es verdad: normalmente existe por lo menos una capa de software entre ellos. Esta capa se llama software intermedio (*middleware*). Como ejemplo de software intermedio consideremos la Figura 28.1. Ésta muestra la comunicación entre un cliente ejecutando un navegador como Internet Explorer y un servidor Web.

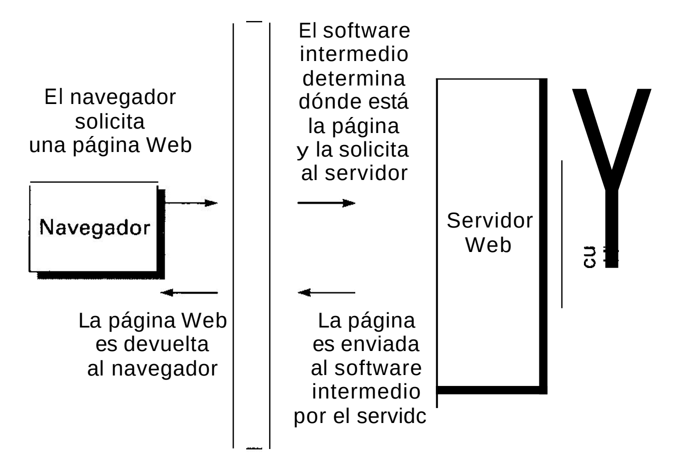


FIGURA 28.1. Software intermedio y servidor Web.

Aquí el software intermedio que se encuentra entre el servidor Web y el cliente que ejecuta el navegador Web intercepta las peticiones que proceden del navegador. Si se hace una petición para una página Web entonces determina la localización del documento Web y envía una petición para esa página. El servidor responde a la petición y devuelve la página al software intermedio, quien la dirige al navegador que la visualizará en la pantalla del monitor que utiliza el cliente. Existen dos categorías de software intermedio: el software intermedio general y el software intermedio de servicios. El primero es el que está asociado a los servicios generales que requieren todos los clientes y servidores. El software típico que se utiliza como tal es:

- El software para llevar a cabo comunicaciones utilizando el protocolo TCP/IP y otros protocolos de red.
- El software del sistema operativo que, por ejemplo, mantiene un almacenamiento distribuido de archivos. Este es una colección de archivos que se espar-

cen por un sistema distribuido. El propósito de esta parte del sistema operativo es asegurar que los usuarios pueden acceder a estos archivos, de forma que no necesiten conocer la localización de los archivos.



El software intermedio es el centro de un sistema cliente/servidor.

- El software de autentificación, el cual comprueba que un usuario que desee utilizar un sistema distribuido pueda en efecto hacerlo.
- El software intermedio orientado a mensajes que gestiona el envío de mensajes desde clientes'a servidores y viceversa.

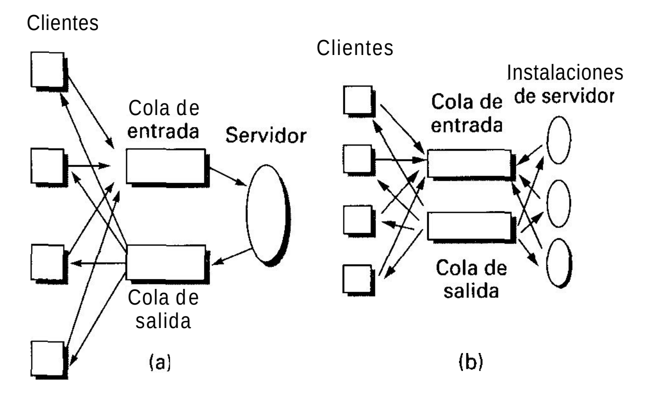


FIGURA 28.2. Configuraciones Mom.

El software intermedio de servicios es el software asociado a un servicio en particular. Entre los ejemplos típicos de este tipo de software se incluyen:

- Un software que permite a bases de datos diferentes conectarse a una red clienteiservidor. Probablemente el mejor ejemplo de este tipo de software sea la ODBC (Conectividad abierta de bases de datos) (Open Database Connectivity)) producida por Microsoft. Esta permite que existan felizmente en una red la vasta mayoría de sistemas de gestión de bases de datos.
- Un software específico de objetos distribuidos tal como el que está asociado a CORBA. CORBA es una tecnología de objetos distribuida que permite **a** objetos escritos en una gran variedad de lenguajes que coexistan felizmente en una red de tal forma que cualquier objeto pueda enviar mensajes a otro objeto. El software intermedio de CORBA tiene que ver con funciones tales como configurar objetos distribuidos, comunicación y seguridad entre objetos.
- Un software intermedio de Web asociado al protocolo HTTP.
- Un software intermedio de software de grupo que administra los archivos que describen a los equipos de trabajo y sus interacciones.
- Un software intermedio asociado a productos de seguridad específicos. Un buen ejemplo es el software intermedio asociado a lo que se conoce como conexiones (*sockets*) seguras. Estas son conexiones que se pueden utilizar para enviar datos seguros en una red de tal forma que hace imposible que intrusos escuchen a escondidas los datos. Este tecnología se describe en la Sección 28.8.

28.2.4. Un ejemplo de software intermedio

El software intermedio orientado a mensajes es el software intermedio que administra el flujo de mensajes hacia y desde un cliente. Esta sección estudia detalladamente este software con el fin de proporcionar un ejemplo de un tipo específico de software intermedio. Para referirse al software intermedio orientado a mensajes a menudo se utiliza la sigla MOM. Es el que gestiona de forma eficaz las colas que contienen mensajes que proceden y que se envían desde servidores. La Figura 28.2 muestra un número de configuraciones. La Figura 28.2 (a) muestra un número de clientes que acceden a dos colas, una cola de entrada a la que envían los mensajes y una cola de salida desde la que toman los mensajes y un único servidor que lee y escribe los mensajes. Un mensaje de entrada típico podría ser el que ordena al servidor que encuentre algunos datos en una base de datos y un mensaje de salida podría ser los datos que se han extraído de la base de datos. La Figura 28.2 (b) muestra múltiples clientes y un número de instanciaciones del programa servidor que trabajan concurrentemente en las mismas colas.

Hay que establecer una serie de puntualizaciones a cerca de un software MOM:

- No se necesita establecer una conexión especializada mientras que el cliente y el servidor interactúan.
- El modelo de interacción entre clientes y servidores es, en efecto, muy simple: los clientes y servidores interactúan mediante colas. Todo lo que necesita hacer el programador del cliente y del servidor es enviar un mensaje a la cola. El software MOM a nivel de programador es muy simple.
- Utilizando el software MOM la comunicación es asíncrona, hasta el punto en que la mitad de la pareja cliente/servidor puede que no esté comunicándose con la otra parte. Por ejemplo, el cliente puede que no esté ejecutándose: puede detenerse para su mantenimiento y que el servidor siga mandando mensajes a colas MOM. Puede que el cliente haya cerrado con algunos mensajes a la espera de que los procese el servidor. El servidor puede colocar estos mensajes en la cola de manera que la próxima vez que el cliente se conecte pueda leer los mensajes. Este escenario es el ideal para la informática móvil.



Cuando se utiliza el software MOM la comunicación es asíncrona.

28.3. ARQUITECTURAS ESTRATIFICADAS

El propósito de esta sección es explorar la idea de una arquitectura estratificada para una aplicación cliente/servidor y se limita a describir las arquitecturas populares de dos y de tres capas. Una arquitectura de dos capas de una aplicación cliente/servidor consiste en una capa de lógica y presentación, y otra capa de bases de datos. La primera tiene que ver con presentar al usuario conjuntos de objetos visuales y llevar a cabo el procesamiento que requieren los datos producidos por el usuario y los devueltos por el servidor. Por ejemplo, esta capa contendría el código que monitoriza las acciones de pulsar botones, el envío de datos al servidor, y cualquier cálculo local necesario para la aplicación. Estos datos se pueden almacenar en una base de datos convencional, en un archivo simple o pueden ser incluso los datos que están en la memoria. Esta capa reside en el servidor.

Normalmente las arquitecturas de dos capas se utilizan cuando se requiere mucho procesamiento de datos. La arquitectura del servidor Web de navegador es un buen ejemplo de una arquitectura de dos capas. El navegador del cliente reside en la capa de lógica y presentación mientras que los datos del servidor Web —las páginas Web— residen en las capa de la base de datos. Otro ejemplo de aplicación en donde se emplearía normalmente una arquitectura de dos capas es una aplicación simple de entrada de datos, donde las funciones principales que ejercitan los usuarios es introducir los datos de formas muy diversas en una base de datos remota; por ejemplo, una aplicación de entrada de datos de un sistema bancario, donde las cuentas de usuarios nuevos se teclean en una base de datos central de cuentas. El cliente de esta aplicación residirá en la capa lógica y de presentación mientras que la base de datos de cuentas residiría en la capa de base de datos.

Las dos aplicaciones descritas anteriormente muestran la característica principal que distingue a las aplicaciones de capas de otras aplicaciones que emplean más capas por el hecho de que la cantidad de procesamiento necesario es muy pequeña. Por ejemplo, en la aplicación de validación de datos, el Único procesamiento requerido del lado del cliente sería la validación de datos que se llevaría a cabo sin recurrir a la base de datos de las cuentas. Un ejemplo sería comprobar que el nombre de un cliente no contiene ningún dígito. El resto de la validación se haría en la capa de la base de datos y conllevaría comprobar la base de datos para asegurar que no se añadieron registros duplicados de clientes a la base de datos. Reece [REE97] ha documentado los criterios que se deberían utilizar cuando consideramos adoptar una arquitectura cliente servidor de dos capas. Estos son los criterios:

- ¿Utiliza la aplicación una única base de datos?
- ¿Se localiza el procesador de base de datos en una sola CPU?
- ¿Es relativamente estática la base de datos en lo que se refiere a predecir un crecimiento pequeño de tamaño y estructura?
- ¿Es estática la base del usuario?
- ¿Existen requisitos fijos o no hay muchas posibilidades de cambio durante la vida del sistema?
- ¿Necesitará el sistema un mantenimiento mínimo?

Aunque algunas de estas cuestiones van enlazadas (los cambios en los requisitos dan lugar a las tareas de mantenimiento), se trata de un cantidad bien amplia de preguntas que deberían de tenerse en cuenta antes de adoptar una arquitectura de dos capas.

CLAVE

Cuando una aplicación implica un procesamiento considerable el enfoque de dos copas cada vez es más difícil de **implementar.**

Cuando una aplicación implica un procesamiento considerable, entonces comienzan a surgir problemas con la arquitectura de dos capas, particularmente aquellas aplicaciones que experimentan cambios de funcionalidad a medida que se van utilizando. También, una arquitectura de dos capas, donde no hay muchas partes de código de procesamiento unidas a acciones tales como pulsación de botones o introducción de texto en un campo de texto, está altamente orientada a sucesos específicos y no a los datos subyacentes de una aplicación, y de aquí que la reutilización sea algo complicado.

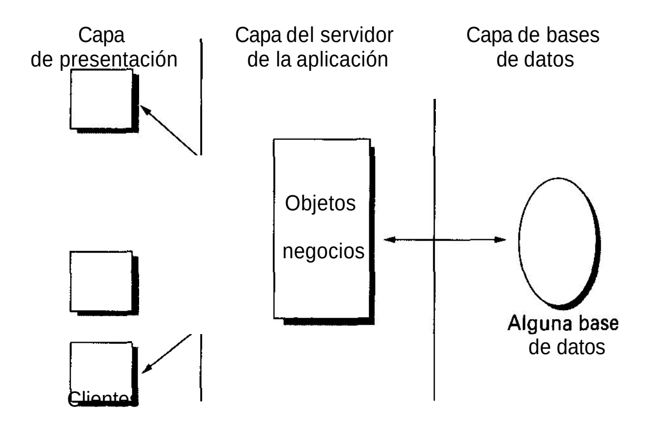


FIGURA **28.3.** Una arquitectura de tres capas.

La Figura 28.3 muestra una arquitectura de tres capas. Se compone de una capa de presentación, una capa de procesamiento (o capa de servidor de solicitudes) y una capa de base de datos. La capa de presentación es la responsable de la presentación visual de la aplicación, la capa de la base de datos contiene los datos de la apli-

cación y la capa de procesamiento es la responsable del procesamiento que tiene lugar en la aplicación. Por ejemplo, en una aplicación bancaria el código de la capa de presentación se relacionaría simplemente con la monitorización de sucesos y con el envío de datos a la capa de procesamiento. Esta capa intermedia contendría los objetos que se corresponden con las entidades de la aplicación; por ejemplo, en una aplicación bancaria los objetos típicos serían los bancos, el cliente, las cuentas **y** las transacciones.

La capa final sería la capa de la base de datos. Ésta estaría compuesta de los archivos que contienen los datos de la aplicación. La capa intermedia es la que conlleva capacidad de mantenimiento y de reutilización. Contendrá objetos definidos por clases reutilizables que se pueden utilizar una y otra vez en otras aplicaciones. Estos objetos se suelen llamar objetos de negocios y son los que contienen la gama normal de constructores, métodos para establecer y obtener variables, métodos que llevan a cabo cálculos **y** métodos, normalmente privados, en comunicación con la capa de la base de datos. La capa de presentación enviará mensajes a los objetos de esta capa intermedia, la cual o bien responderá entonces directamente o mantendrá un diálogo con la capa de la base de datos, la cual proporcionará los datos que se mandarían como respuesta a la capa de presentación.

El lugar donde va a residir la capa intermedia depende de la decisión sobre el diseño. Podría residir en el servidor, que contiene la capa de base de datos; por otro lado, podría residir en un servidor separado. La decisión sobre dónde colocar esta capa dependerá de las decisiones sobre diseño que se apliquen, dependiendo de factores tales como la cantidad de carga que tiene un servidor en particular y la distancia a la que se encuentra el servidor de los clientes. La localización de esta capa no le resta valor a las ventajas que proporciona una arquitectura de tres capas:

- En primer lugar, la arquitectura de tres capas permite aislar a la tecnología que implementa la base de datos, de forma que sea fácil cambiar esta tecnología. Por ejemplo, una aplicación podría utilizar en primer lugar la tecnología relaciona1para la capa de base de datos; si una base de datos basada en objetos funciona tan eficientemente como la tecnología relacional que se pudiera entonces integrar fácilmente, todo lo que se necesitaría cambiar serían los métodos para comunicarse con la base de datos.
- Utiliza mucho código lejos del cliente. Los clientes que contienen mucho código se conocen como clientes pesados (gruesos). En un entorno en donde se suele necesitar algún cambio, estos clientes pesados pueden convertirse en una pesadilla de mantenimiento. Cada vez que se requiere un cambio la organización tiene que asegurarse de que se ha descargado el código correcto a cada cliente. Con una capa intermedia una gran parte del código de una aplicación cliente/servidor reside en un lugar (o en un número reducido pequeño de lugares si se utilizan servidores de copias de seguridad) **y** los cambios de mantenimiento ocurren de forma centralizada.
- La idea de las tres capas encaja con las prácticas orientadas a objetos de hoy en día: todo el procesamiento tiene lugar por medio de los mensajes que se envían a los objetos y no mediante trozos de código asociados a cada objeto en la capa de presentación que se está ejecutando.

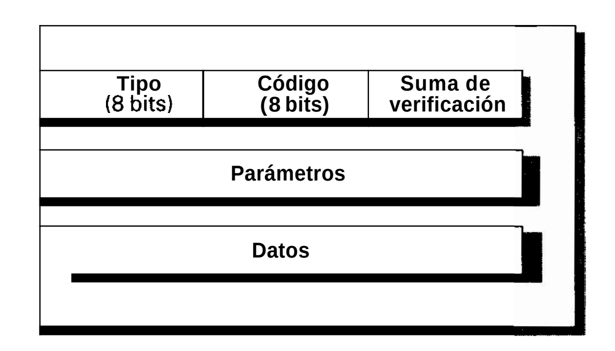


FIGURA 28.4. El formato **del** protocolo ICMP.



Existe una diferencia enire **el** software intermedio (middleware) **y la** capa del servidor de **la** aplicación.

Lo que merece la pena destacar, llegado este punto, es que existe una diferencia entre la capa intermedia del servidor de la aplicación **y** el software intermedio (*middleware*). Mientras que el primero describe el software de la *aplicación* que media entre el cliente y el servidor, el segundo se reserva para el software de sistemas.

28.4 PROTOCOLOS

En este capítulo hasta ahora se ha utilizado el término protocolo sin proporcionar realmente mucho detalle. El propósito de esta sección es proporcionar este detalle.

28.4.1. El concepto

Con objeto de describir lo que es exactamente un protocolo, utilizaré un pequeño ejemplo: el de un cliente que proporciona servicios bancarios para casa, permi-

tiendo, por ejemplo, que un cliente consulte una cuenta cuyos datos residen en el servidor. Supongamos que el cliente se comunica con el servidor que utiliza una serie de mensajes que distinguen las funciones requeridas por el cliente, y que también se comunica con otros datos requeridos por el servidor.

Por ejemplo, el cliente puede ejercer la función de consultar un saldo de cuenta tecleando un número de cuenta en un cuadro de texto y pulsando el botón del ratón el cual enviará el mensaje al servidor. Este mensaje podría ser de la forma

```
CS*<Número de cuenta,
```

donde *CS* (Consulta del Saldo) especifica que el usuario ha consultado una cuenta para obtener el saldo con el número de cuenta que identifica la cuenta. El servidor entonces recibirá este mensaje y devolverá el saldo; el mensaje que el servidor envía podría tener el formato

```
S*<Cantidad del Saldo>
```

El cliente interpretara este mensaje y visualizara el saldo en algún elemento de salida.

Si el cliente quisiera consultar los números de cuentas de todas las cuentas con las que él o ella está asociado, el mensaje entonces podría ser de la forma

```
CC*
```

donde *CC* (Consulta de Cuenta) solicita las cuentas, y en cuya función ya no se requieren más datos. El servidor podría responder con un mensaje que comenzara con (Información de Cuenta) y que estuviera formado por números terminados con asteriscos. Por ejemplo,

```
IC*2238997*88765"882234
```

los mensajes que he descrito forman un protocolo sencillo que comunica funcionalidad y datos entre dos entidades (un cliente y un servidor) en la red.



Un punto importante acerca de los protocolos es que siempre que se utiliza un mecanismo para la comunicación en una red existe un protocolo. Por ejemplo, los objetos distribuidos son objetos que se encuentran en localizaciones remotas en una red, y es mediante la utilización de un protocolo la manera en que se envían los mensajes, aunque la programación de estos objetos esté a un nivel superior por debajo de la programación y oculta al programador.

El protocolo que he descrito se asemeja a un juguete y también a un protocolo de aplicaciones. Merece la pena entrar en el estudio de algunos protocolos más reales que estén asociados a los servicios de nivel de sistema.

28.4.2. IP e ICMP

IP, el protocolo que ejecuta Internet es un protocolo muy complicado, y una descripción completa estaría fuera del ámbito de este libro, ya que la descripción de este protocolo necesitaría un libro entero que le hiciera justicia. Debido a esto me centraré en una parte pequeña del protocolo, aunque importante, conocida como ICMP ***(InternetControl Message Protocol).*** Protocolo de mensajes de control de Internet que se utiliza para monitorizar los errores y problemas de la red que utiliza IP. En la Figura 28.4 se muestra el formato de los mensajes que forman parte del protocolo.

El campo Tipo especifica el tipo de error que ha ocurrido. Por ejemplo, este campo indicaría que el destino de un mensaje era inalcanzable. El campo Código contiene la información secundaria que se utiliza para proporcionar una interpretación más detallada del campo Tipo. El campo de *Suma **de verificación*** es utilizado por el software para comprobar que la transmisión del mensaje ICMP se ha llevado a cabo correctamente. Los dos campos restantes contienen los datos que permitirán al software de comprobación localizar el problema originado.

ICPM es utilizado por IP para llevar a cabo el procesamiento básico de errores y también para incrementar la eficacia de la red. Por ejemplo, se podría utilizar para llevar a cabo el cambio de dirección del mensaje si una computadora, que se encontrara en el camino inicial del mensaje, descubriera una ruta mejor.

28.4.3. POP3

POP3 (Post Office Protocol version 3) (Protocolo de Oficina de Correos versión *3)* se utiliza para el envío y la recepción de correo electrónico. Es un protocolo sencillo y es ampliamente utilizado. En esta sección realizaré un estudio de algunas de sus características.



POP3 es un protocolo robusto y muy sencillo. Utilícelo todo lo posible más que otros protocolos

Existen varios protocolos de correo, entre los que se incluyen SMTP, IMAP y diferentes versiones de POP. Probablemente el más complicado de estos sea IMAP, el cual incluye funciones secundarias y un conjunto de mensajes más rico que POP.

El propósito de POP es posibilitar a los usuarios acceder al correo remoto y consta de una serie de comandos que permiten a los programas de usuario interactuar con un servidor de correo POP. El estándar POP3 describe un grupo de mensajes posibles que pueden enviarse al servidor de correo POP3 y el formato de los mensajes es devuelto por el servidor. La Tabla 28.1 muestra un subconjunto pequeño del protocolo POP3.

Mensaje	*Significado*
USER	*Informa al servidor sobre un usuario en particular que está intentando enviar un correo*
PASS	*Proporciona una contraseña para un usuario en particular*
STAT	*Pregunta al servidor cuántos mensajes hay esperando para ser leídos*
DELE	*Borra un mensaje*
RETR	*Recupera un mensaje*

TABLA 28.1. Un subconjunto del POP3.

28.4.4. El protocolo HTTP

Este es uno de los protocolos más importantes que se utilizan dentro de Internet; es el protocolo que rige la comunicación entre un cliente que utiliza un navegador Web tal como Internet Explorer y un servidor Web.

La función principal de un servidor Web es poner a disposición de clientes páginas Web. Estos clientes utilizarán un navegador que les conectará al puerto 80 en el servidor Web; este es el puerto estándar que **se** utiliza para tales servidores. El navegador que está utilizando el cliente .enviará mensajes definidos por el protocolo HTTP y serán interpretados por el servidor que llevará a cabo operaciones tales como devolver una página Web o procesar algún formulario que esté insertado dentro de una página.

A continuación, se muestra un ejemplo que ilustra lo descrito anteriormente

```
GET/index.html HTTP/ 1.1
```

Este es el mensaje que envía un navegador cuando quiere visualizar una página en particular. Este mensaje se puede haber generado de diferentes maneras. Por ejemplo, el usuario puede haber pinchado con el ratón en un determinado enlace en un documento Web que señala a la página solicitada. La línea anterior contiene la palabra reservada *GET*, la cual especificaba que se iba a devolver un archivo, y especificaba también el nombre del archivo que sigue al carácter '/' (index.html) y la versión del protocolo que utiliza el navegador que va a enviar el mensaje.

El servidor también utilizará el protocolo HTTP para enviar mensajes de vuelta al cliente. Por ejemplo, el mensaje

```
HTTP/1.1 200 OK
```

significa que el servidor, que está utilizando HTTP versión 1.1, ha procesado con éxito la petición iniciada por el cliente. El código 200 en este caso es un ejemplo del código de estado que indica éxito.

28.5 UN SISTEMA DE COMERCIO ELECTRÓNICO

28.5.1. ¿Qué es el comercio electrónico?

Para ilustrar la apariencia de un sistema distribuido examinaremos un ejemplo tomado de un área de aplicación conocida como comercio electrónico. En un sentido amplio, el término «comercio electrónico» (Ecommerce) se puede definir como la aplicación de la tecnología de sistemas distribuidos que apoya las operaciones comerciales. **A** continuación, se detallan algunos de dichos sistemas:

- Sistemas para vender algunos artículos o servicios utilizando Internet, donde los clientes interactúan con el sistema que utiliza navegadores. Entre los sistemas típicos de comercio electrónico de esta categoría se incluyen los que venden libros, ropa y CDs.



- Sistemas para simular alguna actividad comercial en tiempo real utilizando tecnología de red. Un buen ejemplo de este tipo de sistemas es una subasta en la red. Un sistema de subasta típico solicitaría artículos a los usuarios de Internet, ubicaría los datos en una página Web y entonces comenzaría la oferta para ese artículo. Normalmente la compañía de subastas especificaría un período de oferta para así darla por finalizada.
- Sistemas que proporcionan algún servicio basado en red para usuarios. Probablemente los más conocidos son los que ofrecen cuentas de correo gratis, donde los ingresos de dicha empresa probablemente procedan de la publicidad en las páginas Web que se utilizan para ese sitio Web. Estas son compañías que mediante un honorario monitorizan su sitio Web y le envían un mensaje, normalmente por correo electrónico o mediante un buscador si han detectado un problema, como el mal funcionamiento del servidor que se utiliza para el sitio Web.
- Sistemas que proporcionan servicios de asesoramiento. Los sistemas típicos de este tipo son los que procesan una descripción de artículos, como un CD, para los que establecerán el mejor precio, después de haber explorado un número de sistemas de venta en la red.
- Sistemas internos que el cliente no ve, pero que dan soporte a más actividades comerciales convencionales. Por ejemplo, un sistema que apoya el suministro de mercancías a un comerciante minorista de la calle.

- Sistemas de publicidad. Muchos de los ingresos del comercio electrónico proceden de la publicidad en línea. Muchas páginas Web asociadas a las aplicaciones de comercio electrónico contendrán pequeños espacios publicitarios conocidos como *banners*. Estos anuncios se pueden «pinchar», conduciendo así al usuario del navegador a un sitio Web el cual normalmente vende algún producto o servicio. Los sistemas de publicidad son una forma particular de sistemas de comercio electrónico que llevan a cabo funciones tales como vender un *banner* (espacio publicitario), monitorizando el éxito de estos anuncios y la administración del pago de los honorarios de publicidad.

Estas forman un conjunto típico de aplicaciones que están bajo el «*banner*» del comercio electrónico. En la parte restante de esta sección observaremos una aplicación típica de comercio electrónico que administra la venta de un artículo. Esto es lo que piensa cualquier persona de una aplicación de comercio electrónico, aunque espero que después de leer este preámbulo este sistema se considerará sólo como un tipo de sistema.



El comercio electrónico no es solamente un comercio minorista de red.

28.5.2. Un sistema típico de comercio electrónico

Con objeto de entender la naturaleza de los sistemas cliente/servidor, merece la pena examinar un ejemplo de una de las áreas del comercio electrónico. Es un sistema para administrar las ventas de una gran librería que tenga una funcionalidad similar a la que exhiben grandes librerías como Blackwells o la filial británica de Amazon. Las funciones típicas que un sistema como éste proporciona son las siguientes:

- *La provisión de servicios de Catálogo*. Cada libro que venda una compañía estará en catálogo y la página Web proporcionará la descripción de ese libro. La información típica que se proporciona sobre un libro es el nombre, autor, editorial y precio.
- *La provisión de servicios de búsqueda.* El usuario de dicho sistema necesitará navegar por el catálogo para decidir si va a comprar un libro. Esta navegación se podría realizar de muchas maneras diferentes: desde navegar secuencialmente desde el primer libro que aparece, hasta navegar utilizando consultas de búsqueda tal como el título de un libro o su ISBN.
- *La provisión de servicios de pedidos.* Cuando un cliente de un sitio de comercio electrónico quiere comprar un libro, el sistema le proporcionará algún servicio para poderlo hacer y, normalmente, mediante tarjetas de crédito. Generalmente cuando el cliente realiza el pedido de un libro, a continuación se le solicitan los datos de la tarjeta de crédito. Algunos sistemas de pedidos suelen quedarse con los datos de las tarjetas de forma permanente de manera que no se requiere que el cliente proporcione los datos cada vez que haga un pedido.
- *La provisión de servicios de seguimiento.* Estos servicios posibilitarán al cliente seguir con el proceso de una compra utilizando su navegador. Las páginas Web personalizadas para el cliente describirán el desarrollo de un pedido: si ya se ha enviado, si está esperando porque el libro no está en stock, y cuál es la fecha en la que se espera enviar el pedido.
- *El procesamiento de revisiones.* Un servicio ofrecido por los sitios de ventas de libros más sofisticados y es el que proporciona el medio por el que los clientes pueden escribir revisiones de los libros a comprar. Estas revisiones se pueden comunicar al vendedor o bien por correo electrónico o por una página Web especializada.
- *La provisión de un servicio de conferencia.* Un servicio de conferencia capacita a un grupo de clientes para interactuar entre ellos enviando mensajes a una conferencia, entendiendo por conferencia algo dedicado a un tema específico tal como, por ejemplo, la última novela de Robert Goddard o el estado de la ficción criminal. Tales servicios no proporcionan ingresos directamente a un vendedor de libros en red. Sin embargo, sí pueden proporcionar información Útil sobre las tendencias en las compras de libros de las que el personal de ventas de una librería pueden sacar provecho.
- *La provisión de noticias o boletines para clientes.* Un servicio popular que se encuentra en muchos sitios de comercio electrónico dedicados a la venta de un producto es el de proporcionar información por medio del correo electrónico. Para el sitio de ventas de libros que se está describiendo en esta sección se incluirían mensajes tal como textos de revisiones, ofertas especiales o mensajes relacionados con un pedido específico, tal como el hecho de haberse despachado.
- *Control y administración del stock.* Este es un conjunto de funciones que están ocultas para el usuario del sistema de ventas de libros pero que son vitales para el sistema. Estas funciones se asocian a las actividades mundanas, tales como hacer pedidos de libros, hacer el seguimiento de los stocks y reordenar y proporcionar información de ventas al personal responsable de los pedidos.
- *Informes financieros.* Estos son de nuevo un conjunto de funciones que están ocultas ante los ojos del usuario del sistema de ventas de libros pero que son vitales. Proporcionan la información para la gestión financiera de la compañía que ejecuta el sistema y proporciona la información de datos tales como las

ventas día a día, anuales y datos más complejos tales como la efectividad de ciertas estrategias de ventas respecto a las ventas de los libros que eran el tema de estas estrategias.

Se puede decir que estas son un conjunto típico de funciones mostradas por el comercio electrónico dedicado a vender productos; en nuestro caso el producto son libros, aunque no hay ninguna razón para no haber elegido, por ejemplo, ropa, CDs, antigüedades, etc., aunque las funciones del sistema variarían un poco; por ejemplo, en un sitio especializado en vender ropa no habría razón por la que implementar las funciones que están conectadas con las revisiones de productos.

Antes de estudiar la arquitectura de dicho sistema merece la pena hacer hincapié en el desarrollo de tales sistemas. Durante el nivel de análisis hay poca diferencia entre el sistema de comercio electrónico y un sistema más convencional, tal como el que depende de los operadores telefónicos para anotar los pedidos y donde el catálogo se imprime y se envía a los clientes de la forma convencional. Ciertamente existirán funciones que no estarán dentro de tales sistemas convencionales, como por ejemplo los que tienen que ver con las revisiones; sin embargo, la mayor parte de las funciones son muy similares, y es posible que se lleven a cabo de forma diferente; por ejemplo, el proceso de obtener los datos de las tarjetas de crédito sería llevado a cabo por un operador y no por una página Web.



CLAVE

En el nivel de análisis hay muy poca diferencia entre los sistemas de comercio electrónico y los convencionales.

En la Figura 28.5 se muestra la arquitectura técnica de un sistema de libros típico. Los componentes de este sistema son:

- *Clientes Web.* Estos ejecutarán un navegador que interactúa con el sistema y que principalmente lleva a cabo funciones de navegar sobre catálogos y hacer pedidos.
- *Un servidor Web.* Este servidor contendrá todas las páginas Web a las que el cliente irá accediendo y se comunicará con el resto del sistema para proporcionar información tal como la disponibilidad de un libro. Normalmente existirá más de un servidor Web disponible para enfrentarse con un fallo del hardware. Si el servidor Web estaba funcionando y se viene abajo, dicho suceso sería extremadamente seno y se correspondería con las puertas de una librería convencional cerrada a los clientes impidiendo su entrada. Debido a las cajas registradoras, los sistemas de correo electrónico que tienden a ser muy críticos para un negocio tendrán hardware reproducido, incluyendo reproducciones de servidores de Web.

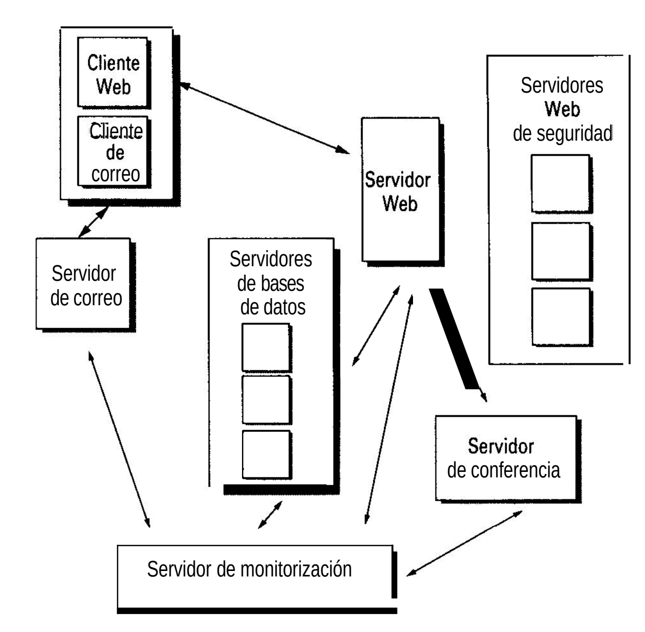


FIGURA 28.5. La arquitectura.

- *Un servidor de correo.* Este servidor mantendrá listas de correos de clientes que, por ejemplo, han indicado que desean estar puntualmente informados de las ofertas especiales y los libros nuevos que se están publicando. Este servidor se comunicará con el servidor Web principal, ya que los clientes proporcionarán sus direcciones de correo y los servicios que quieren, interactuando con las páginas Web que visitan. Esto ilustra un tema importante acerca de los clientes y los servidores: no hay una designación fuerte o rápida de lo que es un cliente o un servidor de un sistema de comercio electrónico: esto depende realmente de la relación entre las entidades implicadas. Por ejemplo, el servidor Web actúa como un servidor para los clientes que ejecutan un navegador, pero actúa como un cliente con el servidor de correo al cual le proporciona direcciones de correo para sus listas de correo.
- *Un servidor de conferencia.* Este es un servidor que administra conferencias. Lee en las contribuciones a la conferencia, visualiza estas contribuciones en una ventana asociada a una conferencia y borra cualquier entidad que no esté actualizada.
- *Servidores de bases de datos.* Son servidores que administran las bases de datos asociadas a la aplicación de comercio. Aquí se incluye la base de datos principal de libros, la cual contiene datos de cada libro; la base de datos de pedidos que contiene datos de los pedidos que han realizado los clientes, todos los que no se han cumplido tanto anteriores como actuales; la base de datos de clientes que contiene datos de los clientes de los vendedores de libros; y una base de datos que contiene datos de las ventas de libros concretos; esta base de datos es particularmente útil para muchos vendedores de libros en

la red, puesto que les permite publicar las listas de los libros más vendidos junto con las ofertas especiales de esos libros. En muchos sistemas de comercio electrónico estas bases de datos son implementadas en varios servidores de bases de datos especializados que son duplicados: tales bases de datos son vitales para el funcionamiento de una compañía de comercio electrónico: si el servidor de bases de datos funcionaba mal y no existen servidores duplicados, ocurría algo muy serio, ya que no tendrían ingresos y los vendedores en red obtendrían una reputación muy pobre. Un punto importante a tener en cuenta sobre esta parte del sistema es que no hay razón para que una tecnología anterior, tal como un *mainframe* que ejecuta una monitorización de transacciones se pueda utilizar para funciones de bases de datos; en realidad muchos de los sistemas de comercio electrónico se componen de un servidor Web frontal que se comunica con un sistema servidor no cliente. Este sistema servidor no cliente es el que lleva existiendo ya desde hace algún tiempo y se utilizaba para un procesamiento más convencional como es el de tomar pedidos por teléfono. Los servidores de bases de datos se mantendrán actualizados por medio de un software de sistemas que detecta cuando se va a aplicar una transacción a una base de datos: primero aplica la transacción a la base de datos que se está utilizando en ese momento y, a continuación, la aplica a todas las bases de datos duplicadas.

- *Un servidor de monitorización.* Este es el servidor que se utiliza para monitorizar la ejecución del sistema. Es utilizado por un administrador del sistema para comprobar el funcionamiento correcto del sistema y también para ajustar el sistema de manera que se archive un rendimiento óptimo.

28.6 TECNOLOGÍAS USADAS PARA EL COMERCIO ELECTRÓNICO

Existen varias tecnologías basadas en red que se utilizan para las aplicaciones de comercio electrónico. Antes de describirlas merece la pena decir que muchas tecnologías antiguas todavía se utilizan para este tipo de aplicación; el mejor ejemplo es el uso de la tecnología de bases de datos relacionales para proporcionar almacenes de datos a gran escala.

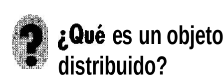


Muchas tecnologías antiguas todavía se utilizan para aplicaciones de comercio electrónico.

28.6.1. Conexiones *(sockets)*

Un *socket* es un tipo de conducto que se utiliza para conectarse a una computadora conectada a una red y basada en TCP/IP. El *socket* se configura de tal manera que los datos pueden ser bajados desde el cliente y devueltos al mismo. Los lenguajes de programación modernos, como Java, proporcionan servicios de alta calidad por medio de los cuales un *socket* se puede conectar mediante «programación» a una computadora cuya dirección de Internet sea conocida y donde los datos se puedan enviar por este conducto. La programación necesaria para esto normalmente no es más complicada que la programación que se requiere para escribir y leer datos de un archivo. Los *sockets* son una implementación a bajo nivel de la conectividad; dentro de las utilizaciones típicas de un *socket* están las aplicaciones de conferencia, donde una entrada a una conferencia se enviaría al servidor de conferencia que utiliza una configuración de *sockets* en el servidor. Los *sockets* son un mecanismo de bajo nivel, pero una forma muy eficiente de comunicar datos en un sistema distribuido que ejecuta el protocolo TCP/IP.

28.6.2. Objetos distribuidos



Un objeto distribuido es aquel que reside en una computadora, normalmente un servidor, en un sistema distribuido. Otras computadoras del sistema pueden enviar mensajes a este objeto como si residiera en su propia computadora. El software del sistema se hará cargo de varias funciones: localizar el objeto, recoger los datos que se requieren para el mensaje y enviarlos a través del medio de comunicación que se utiliza para el sistema. Los objetos distribuidos representan un nivel más alto de abstracción que las conexiones *(sockets)*: a parte de algún código de inicialización, el programador no es consciente del hecho de que el objeto reside en otra computadora. Actualmente existen tres tecnologías de objetos distribuidos en pugna:

- ***RMI.*** Esta es la tecnología asociada al lenguaje de programación de Java. Es un enfoque Java puro en el que solo los programas escritos en ese lenguaje se pueden comunicar con un objeto RMI distribuido. Es la tecnología ideal para sistemas cerrados de Java; estos sistemas generalmente tendrán pocas conexiones o ninguna con otros sistemas.
- *DCOM.* Esta es una tecnología desarrollada por la compañía Microsoft y permite que programas escritos en lenguajes tales como Visual Basic y Visual J++ (la variedad de Java desarrollada por Microsoft) se comuniquen con los objetos que están en computadoras remotas.

- *CORBA*. Esta es la tecnología de objetos distribuidos más sofisticada. Fue desarrollada por un consorcio de compañías informáticas, clientes y compañías de software. La característica más importante del enfoque CORBA es que es multilenguaje, donde los programadores pueden utilizar diferentes lenguajes de programación para enviar mensajes a objetos CORBA: las interfaces CORBA existen para lenguajes tales como Java, FORTRAN, LISP, Ada y Smalltalk. CORBA está al principio de su desarrollo, sin embargo, amenaza con convertirse en la tecnología dominante para objetos distribuidos.

La principal ventaja de los objetos distribuidos sobre la de los *sockets* es el hecho de que como abarca enteramente el paradigma de orientación a objetos se puede emplear los mismos métodos de análisis y diseño que se utilizan para la tecnología de objetos convencional.

28.6.3. Espacios

Esta es una tecnología que se encuentra en un nivel de abstracción incluso más alto que los objetos distribuidos. Fue desarrollada por David Gelemter, un profesor de la Universidad de Yale. La tecnología de espacios concibe un sistema distribuido en base a un gran almacén de datos persistentes donde las computadoras de un sistema distribuido puede leer o escribir. No concibe el sistema distribuido como una serie de programas que pasan mensajes a los demás utilizando un mecanismo como los *sockets,* o como una serie de objetos distribuidos que se comunican utilizando métodos. Por el contrario, la tecnología de espacios conlleva procesos como escribir, leer o copiar datos a partir de un almacén persistente. Un programador que utiliza esta tecnología no se preocupa por detalles como dónde están almacenados los datos, qué proceso va a recoger los datos y cuándo los va a recoger.

Esta tecnología se encuentra sólo al principio, aunque las implementaciones llevan ya disponibles durante algún tiempo para lenguajes como C y C++; sin embargo, la implementación de la tecnología dentro de Java como Javaspaces debería asegurar que cada vez se utilizará más para las aplicaciones principales.

28.6.4. CGI


¿Para qué se utiliza CGI?

El término CGI *(Common Gateway Interface)* significa Interfaz común de pasarela. Es la interfaz con el servidor Web al cual se puede acceder mediante los programas que se ejecutan en el servidor. Gran parte de la interactividad asociada a las páginas Web se implementa programando el acceso a la CGI. Por ejemplo, cuando el usuario de un navegador accede a una página que contiene un formulario, éste lo rellena y lo envía al servidor Web, programa que accede al CGI, procesa el formulario, y lleva a cabo la funcionalidad asociada al formulario; por ejemplo, recuperando los datos solicitados en el formulario. La programación CGI ***se*** puede llevar a cabo en varios lenguajes de programación, aunque el lenguaje seleccionado ha sido Perl, lenguaje de procesamiento de cadenas, existen otros entre los que se incluyen, por ejemplo, Java y C**++**, que contienen los servicios para el procesamiento CGI. Recientemente los que desarrollan Java han proporcionado a los programadores el servicio de utilizar Java para este tipo de programación que utiliza la tecnología conocida como *sewlets*. Estos trozos pequeños de código Java son insertados en un servidor de Web y se ejecutan cuando ocurre un suceso, como enviar un formulario. Los servlets ofrecen un alto grado de portabilidad sobre otros lenguajes de programación.

28.6.5. Contenido ejecutable

Este es el término que se aplica a la inclusión en una página Web de un programa que se ejecuta cuando la página es recuperada por un navegador. Este programa puede llevar a cabo un número diverso de funciones entre las que se incluyen la animación y la presentación de un formulario al usuario para insertar datos. Existen varias tecnologías que proporcionan servicios para insertar contenido ejecutable en una página Web. Aquí se incluyen *applets,* Active X y Javascript. Un *applet* es un programa escrito en Java que interactúa con una página Web. Lo importante a señalar de esta tecnología es que es portátil: el código Java se puede mover fácilmente desde un sistema operativo a otro, pero es potencialmente inseguro. La razón de por qué los *applets* son inseguros es que se pueden utilizar como medio de transmisión de virus y otros mecanismos de acceso a una computadora. Cuando una página de navegador que contiene un *applet* es descargada por un navegador, es lo mismo que cargar un programa en la computadora cliente que ejecuta el navegador. Afortunadamente los diseñadores de Java desarrollaron el lenguaje de tal manera que es inmensamente difícil escribir *applets* que provoquen violaciones en la seguridad. Desafortunadamentela solución que se adoptó evita acceder a los recursos de una computadora cliente o ejecutar otro programa en la computadora. Aunque se han hecho muchas mejoras en los *applets* que permiten un acceso limitado, el modelo principal del uso de *applets* está restringido a este modo de ejecución.

Active X es otra tecnología de contenido ejecutable que fue desarrollada por Microsoft. De nuevo se trata de un código de programa insertado en una página Web; la diferencia principal entre esta tecnología y los *applets* es el hecho de que estos trozos de código se pueden escribir en diferentes lenguajes como Visual Basic y C++. Esta forma de contenido ejecutable también sufre de posibles problemas de seguridad.

Javascript es un lenguaje de programación interpretado y sencillo que se inserta directamente en una página Web. Se diferencia de las soluciones de Active X y

applets en que el código fuente de cualquier programa de guiones de Java se integra en una página Web en vez de en el código de objetos como ocurre con los *applets* y Active X. Javascript es un lenguaje sencillo que se utiliza para una programación relativamente a bajo nivel.

28.6.6. Paquetes cliente/servidor

Este término describe las colecciones de software que normalmente llevan a cabo algún tipo de procesamiento de sistemas. A continuación, se muestra un grupo de ejemplos típicos de paquetes de software:

- *Paquetes de reproducción de datos.* Este tipo de software realiza una transacción en la base de datos y la aplica a un número de bases de datos reproducidas, evitando así acceder a estas bases de datos hasta que estén todas en sincronización.
- *Paquetes de seguridad.* Estos son paquetes que monitorizan el tráfico dentro de un sistema distribuido y avisan al administrador de sistemas de la aparición de cualquier violación posible en la seguridad. Por ejemplo, el hecho de que alguien intente entrar en un sistema con una contraseña sin reconocer.
- *Monitores de transacciones.* Estos son paquetes de software que administran las transacciones que tienen lugar dentro de un sistema distribuido y aseguran que se devuelvan los datos correctos como resultado de una transacción y en el orden correcto. Muchas de las funciones de estos monitores tienen que ver con asegurar que los resultados correctos se devuelvan incluso en el entorno en donde podrían aparecer errores de hardware o de transmisión.

28.7 EL DISEÑO DE SISTEMAS DISTRIBUIDOS

Antes de observar algunos de los principios del diseño que se utilizan en el desarrollo de sistemas distribuidos, particularmente de sistemas de comercio electrónico, merece la pena reiterar lo que se ha señalado anteriormente: a nivel de análisis hay poca diferencia entre un sistema distribuido y un sistema local, y se basa en que el modelo de análisis de un sistema no contendrá ningún dato de diseño como el hecho de que tres computadoras, y no una solo, están llevando a cabo algún procesamiento. Esto significa que el desarrollador de un sistema distribuido se enfrentará normalmente con un modelo de objetos o un modelo funcional similar al que se mostró en las primeras partes de este libro; esta descripción irá acompañada de algunos de los tipos de computadoras y de hardware de redes que se van a utilizar. El proceso de diseño implica transformar el modelo de análisis en algún modelo físico que se implementa en los elementos de hardware del sistema.



Recuerde que durante el análisis existe poca diferencia entre una aplicación de comercio electrónico y una convencional.

Es necesario que el diseñador de sistemas distribuidos conozca una serie de principios de diseño. El resto de esta sección muestra un esquema de los mismos.

28.7.1. Correspondencia del volumen de transmisión con los medios de transmisión

Este es uno de los principios más obvios. Esto significa que para un tráfico denso de datos en un sistema distribuido se deberían utilizar medios de transmisión rápidos (y caros). El proceso de asignar tales medios normalmente tiene lugar después de haber tomado decisiones sobre la potencia de procesamiento de distribución en un sistema y, algunas veces, conlleva unas ligeras iteraciones al final de la fase de diseño.

28.7.2. Mantenimiento de los datos más usados en un almacenamiento rápido

Este principio también es obvio. Requiere que el diseñador examine los patrones de datos en un sistema y asegure que los datos a los que se accede frecuentemente se guarden en algún medio de almacenamiento rápido. En muchos sistemas tales datos pueden constituir no más del 5 por 100 de los datos originales almacenados en el sistema, y de esta manera permite utilizar con frecuencia las estrategias que conllevan el almacenamiento de estos datos dentro de la memoria principal.

28.7.3. Mantenimiento de los datos cerca de donde se utilizan

Este principio de diseño intenta reducir el tiempo que pasan los datos en medios lentos de transmisión. Muchos de estos sistemas son en donde los usuarios acceden con frecuencia a un subconjunto de datos. Por ejemplo, un sistema distribuido usado en una aplicación bancaria contendría bases de datos con datos de las cuentas de los clientes, en donde la mayor parte de las consultas a las bases de datos de las sucursales las realizarán los clientes de esa sucursal; entonces, si los datos de un sistema bancario se distribuyen a los servidores de las sucursales, y los datos asociados a los clientes de esa sucursal están en esa sucursal, el resto de los datos podrían estar en otros servidores con otras ubicaciones, y cualquier consulta que se originara sobre los datos se tendría que comunicar a través de líneas lentas de transmisión.

28.7.4. Utilización de la duplicación de datos todo lo posible

La duplicación consiste en mantener múltiples copias de datos en un sistema al mismo tiempo. Existen muchas razones para la duplicación de datos. La primera es que hay que asegurar la redundancia que permite que un sistema distribuido continúe funcionando aun cuando una computadora con datos importantes quede fuera de servicio normalmente por un mal funcionamiento del hardware. La otra razón es que proporciona una forma de implementar el principio dilucidado en la sección anterior: el de asegurar que los datos estén ubicados cerca de donde se utilizan. Por ejemplo, una compañía hotelera con una base central de reservas que hace el seguimiento de todas las reservas de las habitaciones para todos sus hoteles. Es posible que esta compañía tenga dos puntos de contacto para clientes que deseen hacer las reservas: los hoteles en **sí** y una oficina central de reservas. Una forma de asegurar el alto rendimiento es duplicando los datos asociados a un hotel en particular y guardar los datos en el servidor ubicado en el hotel. Esto significa que cualquier reserva realizada por el hotel solo necesitará acceder a una base de datos local y no requerirá ningún tráfico en la línea lenta de transmisión. Esto suena a un principio muy simple de implementación sencilla. Desafortunadamente, las cosas nunca son tan simples. En el ejemplo de las reservas de hoteles no hay necesidad de que las bases de datos asociadas a los hoteles se comuniquen con la base de datos central. La razón que apoya esto es el hecho de que también habrá clientes que utilicen la oficina central de reservas, así como clientes realizando reservas de habitaciones que ofrecen los mismos hoteles. **A** menos que haya coordinación entre la base de datos de la oficina central de reservas **y** las bases de datos individuales duplicadas de cada hotel, surgirán problemas: por ejemplo, el hecho de decir que hay una habitación libre en un momento concreto en un hotel a un cliente que utiliza el servicio central de reservas aun cuando esa habitación ya ha sido reservada por otro cliente que ha llamado directamente al hotel.



La duplicación controlada de datos puede tener un efecto importante en el rendimiento de un sistema distribuido.

El problema anterior implica que en un sistema donde hay una relación dinámica entre las bases de datos individuales existe la necesidad entonces de que cada base de datos mantenga informadas de los cambios a otras bases de datos, y de que aseguren que los cambios se reflejen en todos los datos duplicados. Esto también implica la aparición de retrasos porque las transacciones permanecerán en cola esperando a que la base de datos se sincronice con otras bases de datos. Esto no significa que se tenga que utilizar la duplicación de datos, lo que significa es que se necesita un diseño cuidadoso para minimizar la cantidad de gastos asociados a él en las transmisiones.

Hay que señalar que en los sistemas distribuidos donde existe menos relación dinámica entre los datos, se pueden emplear estrategias más simples que eliminen muchos de los gastos. Por ejemplo, un banco normalmente lleva a cabo transacciones una vez al día en las bases de datos de los clientes y normalmente después del cierre del negocio. Esto significa que un sistema bancario distribuido puede duplicar datos en sus sucursales y solamente puede volver a copiar los datos cambiados en las bases de datos una vez al día: no habría necesidad de coordinar las bases de datos con frecuencia durante el día de trabajo.

Habría que señalar también que esta subsección ha tratado la duplicación de datos en función de mantener los datos cerca de los usuarios para reducir el tiempo de transmisión con medios lentos. Hay otras razones para duplicar los datos, por ejemplo una base de datos que se utiliza mucho tendrá colas de transacciones preparadas y esperando a ejecutarse. Estas colas se pueden reducir disfrutando de bases de datos idénticas mantenidas en otros servidores de bases de datos concurrentes.

28.7.5. Eliminar cuellos de botella

En un sistema distribuido un servidor se convierte con frecuencia en un cuello de botella: tiene que manipular tanto tráfico que se construyen grandes colas de transacciones esperando a ejecutarse, con el resultado de que los servidores que están esperando los resultados del procesamiento estarán, en el mejor de los casos, ligeramente cargados y, en el peor, inactivos. La estrategia normal para manipular cuellos de botella es compartir la carga de procesamiento entre los servidores, normalmente servidores físicamente cerca del que está sobrecargado.

28.7.6. Minimizar la necesidad de un gran conocimiento del sistema

Los sistemas distribuidos suelen necesitar conocer el estado del sistema completo, por ejemplo, podría ser que necesitaran conocer la cantidad de registros de una base de datos central. El hecho de necesitar este conocimiento genera más tráfico reduciendo así la eficiencia de un sistema, ya que generará tráfico extra a lo largo de las líneas de transmisión. El diseñador de un sistema distribuido en primer lugar necesita minimizar que el sistema dependa de datos globales, y entonces asegurar que el conocimiento necesario se comunique rápidamente a aquellos componentes del sistema que lo requieran.

28.7.7. Agrupar datos afines en la misma ubicación

Los datos que están relacionados deberían de estar dentro del mismo servidor. Por ejemplo, en una aplicación de reservas en un sistema de vacaciones, los paquetes

individuales de vacaciones deberían de estar cerca de los datos que describen las reservas actuales de ese paquete. Ubicar por separado los datos en diferentes servidores asegura que los medios de baja transmisión y muy cargados se cargarán incluso más. El diseñador de un sistema distribuido debe asegurarse de que los datos relacionados gracias al hecho de que se suelen recuperar juntos tendrán que residir lo más cerca posible, preferiblemente en el mismo servidor, o si no, y no de manera tan preferible, en servidores conectados a través de medios de transmisión rápidos tales como los medios utilizados en una red de área local.

28.7.8. Considerar la utilización de servidores dedicados a funciones frecuentes

Algunas veces se puede lograr un mayor rendimiento mediante la utilización de un servidor de empleo específico para una función en particular en lugar de, por ejemplo, un servidor de bases de datos.

28.7.9. Correspondencia de la tecnología con las exigencias de rendimiento

Muchas de las tecnologías que se estudian en este capítulo tienen pros y contras, y un factor importante aquí son las demandas de rendimiento de una tecnología en particular.

Por ejemplo, como medio de comunicación, las conexiones (sockets) normalmente son un medio de comunicación mucho más rápido que los objetos distribuidos. Cuando el diseñador elige una tecnología debe de tener conocimiento de la transmisión y de las cargas de procesamiento que conlleva, y seleccionar una tecnología que minimice estas cargas.

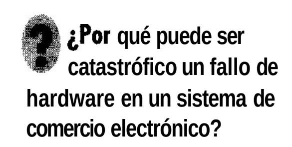
28.7.10. Empleo del paralelismo todo lo posible

Una de las ventajas principales de la tecnología cliente/servidor es el hecho de que se pueden añadir servidores y, hasta cierto punto, elevar el rendimiento del sistema. Muchas funciones del comercio electrónico pueden beneficiarse de la ejecución que están llevando a cabo diferentes servidores en paralelo. Esta no es una decisión sencilla. Mediante el empleo de varios servidores, el diseñador está creando la necesidad de que estos servidores se comuniquen, por ejemplo, un servidor puede que necesite a otro para completar una tarea en particular antes de finalizar la suya propia. Esta comunicación puede introducir retrasos y, si el diseñador no tiene cuidado, pueden negar los avances de rendimiento que se han logrado utilizando el paralelismo.

28.7.11. Utilización de la compresión de datos todo lo posible

Se dispone de un grupo de algoritmos que comprimen datos y que reducen el tiempo que tardan los datos en transferirse entre un componente de un sistema distribuido y otro componente. El único gasto que se requiere para utilizar esta técnica es tiempo del procesador y la memoria necesaria para llevar a cabo la compresión en la computadora del emisor y la descompresión en la computadora del receptor.

28.7.12. Diseño para el fallo



¿Por qué puede ser catastrófico un fallo de hardware en un sistema de comercio electrónico?

Un fallo de hardware en la mayoría de los sistemas de comercio electrónico es una catástrofe: para los sistemas de ventas en red esto es equivalente a echar el cierre a los clientes. Una parte importante del proceso de diseño es analizar los fallos que podrían aparecer en un sistema distribuido y diseñar el sistema con suficiente redundancia como para que dicho fallo no afecte seriamente y, en el mejor de los casos, que se pueda reducir el tiempo de respuesta de ciertas transacciones. Una decisión normal que suele tomar el diseñador es duplicar los servidores vitales para el funcionamiento de un sistema distribuido. Una estrategia en los sistemas de alta integridad es que un servidor se reproduzca tres veces y que cada servidor lleve a cabo la misma tarea en paralelo. Cada servidor produce un resultado a comparar. Si los tres servidores aceptan el resultado, éste pasa a cualquier usuario o servidor que lo requiera; si uno de los servidores no está de acuerdo, entonces surge un problema y el resultado de la mayoría se pasa informando al administrador de sistemas del posible problema. La duplicación de servidores como estrategia de mitigación de fallos puede utilizarse junto con el diseño de un sistema para lograr el paralelismo en las tareas.

28.7.13. Minimizar la latencia

Cuando los datos fluyen de una computadora a otra en un sistema distribuido a menudo tiene que atravesar otras computadoras. Algunas de estas computadoras puede que ya tengan unos datos que expidan funcionalidad; es posible que otras procesen los datos de alguna manera. El tiempo que tardan las computadoras es lo que se conoce como latencia. Un buen diseño de sistema distribuido es el que minimiza el número de computadoras intermediarias.

28.7.14. Epílogo

Este ha sido un estudio breve aunque necesario, y se han tratado las diferentes estrategias de diseño que se utilizan en los sistemas distribuidos que implementan las funciones del comercio electrónico. Un punto importante a tener en cuenta antes de abandonar esta sección

es que una estrategia puede militar contra otra, minimizar la latencia y duplicar bases de datos pueden ser dos estrategias opuestas: incrementar el número de bases de datos duplicadas incrementa la latencia de un sistema; como consecuencia, el diseño de sistemas distribuidos, más que otra cosa, es un arte.

28.8 INGENIERÍA DE SEGURIDAD

El incremento masivo en la utilización pública de sistemas distribuidos ha dado lugar a algunos problemas graves con la seguridad. Los sistemas distribuidos anteriores se suelen localizar en un lugar físico utilizando tecnologías tales como redes de área local. Dichos sistemas estaban físicamente aislados de los usuarios externos y como consecuencia la seguridad, aunque es un problema, no era un problema enorme como lo es ahora para los sistemas de comercio electrónico a los que pueden acceder miembros de usuarios que emplean un navegador.

A continuación, se detallan algunas de las intrusiones típicas en la seguridad que pueden ocurrir:

- Un intruso monitoriza el tráfico de una línea de transmisión y recoge la información confidencial que genera un usuario. Por ejemplo, el intruso podría leer el número, la fecha de caducidad y el nombre del titular de una tarjeta de crédito. Y, a continuación, puede utilizar esta información para realizar pedidos en Internet.
- Un intruso podría entrar en un sistema distribuido, acceder a la base de datos y cambiar la información de la misma. Por ejemplo, podría cambiar el balance de una cuenta en un sistema bancario en la red.



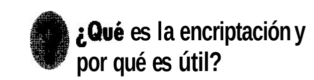
Ahora que Internet ya se estó utilizando para muchas mós aplicaciones, la seguridad se convierte en un problema importante.

- Un intruso podría leer una transacción que pasa por alguna línea de transmisión y alterar los datos dentro de ella en beneficio propio. Por ejemplo, podría alterar la instrucción de un cliente de un sistema bancario en red para transferir los datos de una cuenta a otra para que la cuenta del intruso sea la que tiene lugar en la transferencia.
- Un ex empleado contrariado de una compañía envía un programa al sistema distribuido de la compañía monopolizando el tiempo del procesador del sistema, pasando gradualmente de servidor a servidor hasta que el sistema queda exhausto y acaba parándose. Esta es una forma de ataque conocido como denegación de ataque al servicio.
- Un empleado contrariado de una compañía envía un programa a un sistema distribuido el cual borra los archivos importantes del sistema.

Estas son entonces algunas de las intrusiones que pueden tener lugar dentro de un sistema distribuido; estas intrusiones cada vez son más frecuentes, ya que gran parte de la transmisión en los sistemas de comercio electrónico ocurren en una Internet públicamente accesible que utiliza protocolos públicamente disponibles.

Una de las tareas más importantes del diseñador de sistemas distribuidos es diseñar un sistema con el propósito de minimizar la posibilidad de éxito de las instrucciones de alto riesgo. Para poder llevarlo a cabo se necesita utilizar una serie de tecnologías. En la siguiente sección se hace una relación detallada de estas tecnologías.

28.8.1. Encriptación



Encriptación es el término que se utiliza para referirse al proceso de transformar datos o texto (texto en claro) para ser ilegible; además, debería resultar virtualmente imposible que un intruso pueda descifrarlo. **A** continuación, se detalla el proceso de utilización de esta tecnología:

1. La computadora emisora transforma el texto en alguna forma ilegible; este proceso se conoce como encriptación.
2. Los datos encriptados entonces se envían a través de líneas de transmisión insegura.
3. La computadora receptora procesa el texto encriptado y lo transforma a su forma original. Este proceso se conoce como desencriptación.

Se utilizan dos formas de encriptación. La primera es la encriptación simétrica, donde un mensaje se transforma en una forma encriptada utilizando una cadena conocida como clave: se aplica alguna transformación en el mensaje utilizando la clave. Entonces la clave se comunica al receptor a través de algún medio seguro **y** es utilizado por el receptor para llevar a cabo la desencriptación.

La encriptación simétrica es eficiente pero padece un problema importante: si un intruso descubre la clave, podría descubrir fácilmente el mensaje encriptado. La encriptación de clave pública es una solución a este problema, donde se utilizan dos claves de encriptación: una conocida como clave pública **y** otra como clave privada. Un usuario que desea recibir mensajes encriptados publicará **su** clave pública. Otros

usuarios que deseen comunicarse con el usuario utilizarán esta clave para encriptar cualquier mensaje; los mensajes entonces son desencriptados mediante la clave privada cuando los recibe el usuario original. Esta forma de encriptación tiene la ventaja principal de que la gestión de claves es sencilla: la clave privada nunca se envía a nadie. Un intruso que monitoriza los datos encriptados que viajan por algún medio de transmisión es incapaz de decodificar ningún mensaje puesto que no tienen acceso posible a la clave privada. El inconveniente principal de esta forma de encriptación es que se necesita una gran cantidad de recursos para llevar a cabo el proceso de desencriptación. Debido a esta clave pública, los sistemas normalmente se limitan a envíos cortos de texto o a un texto pensado para ser altamente seguro. También se utiliza para la autenticación, la cual utiliza una técnica conocida como firmas digitales, que se describen en la sección siguiente.

La tecnología principal que se utiliza en Internet para la encriptación simétrica es la capa de *sockets* seguros **(SSL).** Esta es una tecnología que se utiliza para encriptar datos confidenciales tales como los números de las tarjetas de crédito que viajan desde el navegador a un servidor Web, o desde una aplicación a otra.

28.8.2. Funciones de compendio de mensajes

Una función de compendio de mensajes es un algoritmo que genera un número grande —normalmente entre 128 y 256 bits de longitud— que representa un compendio o resumen de los caracteres de un mensaje. Tiene la propiedad de que si el mensaje cambia y se vuelve a aplicar el algoritmo, el número entonces cambia. Existen muchas utilizaciones de las funciones de compendio de mensajes. Un uso común es detectar los cambios en un mensaje, por ejemplo, el hecho de que una transacción financiera se haya modificado en la transmisión con objeto de favorecer al que modifica. Antes de enviar un mensaje se aplica la función de compendio de mensaje y se forma un número grande. El mensaje entonces se envía con el número añadido al final. La función de compendio de mensaje se aplica en el receptor y el número resultante se compara con el que se envió. Si los dos son iguales el mensaje entonces no se ha modificado: si no son iguales el mensaje ha sido modificado durante la transmisión.

28.8.3. Firmas digitales

Una firma digital, como su propio nombre sugiere, es una forma de que el emisor de un mensaje se pueda identificar con el receptor de tal manera que el receptor pueda confiar en que el mensaje fue enviado realmente por el emisor. La encriptación de clave pública es la que se utiliza normalmente para esto. Consideremos una forma de llevar a cabo este proceso: dos usuarios de computadoras A y B quieren comunicarse, y A quiere asegurarse de que se está comunicando con B. Para poder hacer esto, B necesita tener una clave pública y una privada, y tener conocimiento de cuál es la clave pública. A envía un mensaje a B con un texto en el que A pide a B una encriptación utilizando su clave privada. Este texto entonces es encriptado por B y devuelto a A quien lo descifra utilizando la clave pública que B le ha proporcionado. Si el mensaje es el mismo que el que se envió, B es entonces quien dice que es, pero, si no lo es, B entonces no ha demostrado su identidad. Este esquema comparte todas las ventajas de cualquier método de clave pública en donde la clave privada es segura; sin embargo, puede ser atacado por cualquiera que desee establecer falta de confianza entre un emisor y un receptor. Esto se puede hacer alterando el mensaje encriptado que se ha devuelto al emisor.

28.8.4. Certificaciones digitales

Una certificación digital es un documento electrónico que proporciona al usuario un alto de grado de confianza con una organización o persona con la que estén tratando. Se pueden utilizar cuatro tipos de certificaciones:

- *Certificaciones de una autoridad de certificación.* Una autoridad de certificación es una organización que proporciona certificados digitales, tales como estas dos organizaciones: Canada Post Corporation y los servicios postales de US.
- *Certificaciones del servidor.* Estas certificaciones contienen datos tales como la clave pública del servidor, el nombre de la organización que posee el servidor y la dirección del servidor en Internet.
- *Certificaciones personales.* Estas son las certificaciones asociadas a un individuo. Contendrán información física tal como la dirección del individuo junto con la información relacionada con la computadora como la clave pública y la dirección de correo electrónico de la persona.
- *Certificaciones del editor de software.* Estos son certificados que proporcionan confianza en que el software ha sido producido por una compañía de software específica.

Como ejemplo para el funcionamiento de estas certificaciones tomemos el de las certificaciones del servidor. Un servidor que utiliza la capa de *sockets* seguros (SSL) debe de tener un certificado SSL. Este certificado contiene una clave pública. Cuando un navegador se conecta con el servidor se utiliza entonces esta clave pública para codificar la interacción inicial entre el servidor y el navegador.

28.9 COMPONENTES DE SOFTWARE PARA SISTEMAS C/S

28.9.1. Introducción

En lugar de visualizar el software como una aplicación monolítica que deberá de implementarse en una máquina, el software que es adecuado para una arquitectura posee varios componentes distintos que se pueden asociar al cliente o al servidor, o se pueden distribuir entre ambas máquinas:

Componente de interacción con el usuario y presentación. Este componente implementa todas las funciones que típicamente se asocian a una interfaz gráfica de usuario (IGU).

Componente de aplicación. Este componente implementa los requisitos definidos en el contexto del dominio en el cual funciona la aplicación. Por ejemplo, una aplicación de negocios podría producir toda una gama de informes impresos basados en entradas numéricas, cálculos, información de una base de datos y otros aspectos. Una aplicación de software de grupo podría proporcionar funciones para hacer posible la comunicación mediante boletines de anuncios electrónicos o de correo electrónico, y en nuestro caso de estudio esto conllevaría la preparación de informes como los que describen las ventas de libros. En ambos casos, el software de la aplicación se puede descomponer de tal modo que alguno de los componentes residan en el cliente y otros residan en el servidor.



la PFR del grupo de noticias **comp.client-server** se puede encontrar en la página
www.faqs.org/faqs/client-server-faq

Componente de gestión de bases de datos. Este componente lleva a cabo la manipulación y gestión de datos por una aplicación. La manipulación y gestión de datos puede ser tan sencilla como la transferencia de un registro, o tan compleja como el procesamiento de sofisticadas transacciones SQL.

Además de estos componentes, existe otro bloque de construcción del software, que suele denominarse software intermedio en todos los sistemas C/S. El *software intermedio* se describió en la Sección 28.2.3. Orfali [ORF99] y sus colegas se han referido al software intermedio como «el sistema nervioso de un sistema cliente/servidor».



El software intermedio establece la infraestructura que hace posible que los componentes **cliente/servidor** interoperen.

28.9.2. Distribución de componentes de software

Una vez que se han determinado los requisitos básicos de una aplicación cliente/servidor, el ingeniero del software debe decidir cómo distribuir los componentes de software descritos en la Sección 28.1.1 entre el cliente y el servidor. Cuando la mayor parte de la funcionalidad asociada a cada uno de los tres componentes se le asocia al servidor, se ha creado un diseño de *servidor pesado* (grueso). A la inversa, cuando el cliente implementa la mayor parte de los componentes de interacción/presentación con el usuario, de aplicación y de bases de datos, se tiene un diseño de *cliente pesado* (grueso).

Los clientes pesados suelen encontrarse cuando se implementan arquitecturas de servidor de archivo y de servidor de base de datos. En este caso el servidor proporciona apoyo para la gestión de datos, pero todo el software de aplicación y de IGU reside en el cliente. Los servidores pesados son los que suelen diseñarse cuando se implementan sistemas de transacciones y de trabajo en grupo. El servidor proporciona el apoyo de la aplicación necesario para responder a transacciones y comunicaciones que provengan de los clientes. El software de cliente se centra en la gestión de IGU y de comunicación.



Un cliente «pesado» implementa la mayoría de las aplicaciones especificas de la aplicación en el cliente. Un cliente «ligero» relega la mayor parte del proceso al servidor.

Se pueden utilizar clientes y servidores pesados para ilustrar el enfoque general de asignación de componentes de software de cliente/servidor. Sin embargo, un enfoque más granular para la asignación de componentes de software define cinco configuraciones diferentes.

Presentación distribuida. En este enfoque cliente/servidor rudimentario, la lógica de la base de datos y la lógica de la aplicación permanecen en el servidor, típicamente en una computadora central. El servidor contiene también la lógica para preparar información de pantalla, empleando un software tal como CICS. Se utiliza un software especial basado en PC para transformar la información de pantalla basada en caracteres que se transmite desde el servidor en una presentación IGU en un PC.

¿Cuáles son las opciones de configuración poro los componentes de software cliente/servidor?

Presentación remota. En esta extensión del enfoque de presentación distribuida, la lógica primaria de la base

de datos y de la aplicación permanecen en el servidor, y los datos enviados por el servidor serán utilizados por el cliente para preparar la presentación del usuario.

Lógica distribuida. Se asignan al cliente todas las tareas de presentación del usuario y también los procesos asociados a la introducción de datos tales como la validación de nivel de campo, la formulación de consultas de servidor y las solicitudes de informaciones de actualizaciones del servidor. Se asignan al servidor las tareas de gestión de las bases de datos, y los procesos para las consultas del cliente, para actualizaciones de archivos del servidor, para control de versión de clientes y para aplicaciones de ámbito general de la empresa.

Gestión de datos remota. Las aplicaciones del servidor crean una nueva fuente de datos dando formato a los datos que se han extraído de algún otro lugar (por ejemplo, de una fuente de nivel corporativo). Las aplicaciones asignadas al cliente se utilizan para explotar los nuevos datos a los que se ha dado formato mediante el servidor. En esta categoría se incluyen los sistemas de apoyo de decisiones.

Bases de datos distribuidas. Los datos de que consta la base de datos se distribuyen entre múltiples clientes y servidores. Consiguientemente, el cliente debe de admitir componentes de software de gestión de datos así como componentes de aplicación y de IGU.

Durante los Últimos años se ha dado mucha importancia a la tecnología de cliente ligero. Un cliente ligero es la llamada «computadorade redes» que relega todo el procesamiento de la aplicación a un servidor pesado. Los clientes ligeros (computadoras de red) ofrecen un coste por unidad sustancialmente más bajo a una pérdida de rendimiento pequeña o nada significativa en comparación con las computadoras de sobremesa.

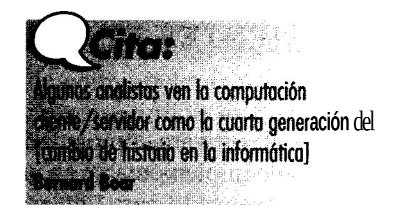
28.9.3. Líneas generales para la distribución de componentes de aplicaciones

Aun cuando no existen reglas absolutas que describan la distribución de componentes de aplicaciones entre el cliente y el servidor, suelen seguirse las siguientes líneas generales:

El componente de presentaciónlinteracción suele ubicarse en el cliente. La disponibilidad de entornos basados en ventanas y de la potencia de cómputo necesaria para una interfaz gráfica de usuario hace que este enfoque sea eficiente en términos de coste.

Si es necesario compartir la base de datos entre múltiples usuarios conectados a través de la LAN, entonces la base de datos suele ubicarse en el servidor. El sistema de gestión de la base de datos y la capacidad de acceso a la base de datos también se asignan al servidor, junto con la base de datos física.

Los datos estáticos que se utilicen deberían de asignarse al cliente. Esto sitúa los datos más próximos al usuario que tiene necesidad de ellos y minimiza un tráfico de red innecesario y la carga del servidor.



El resto de los componentes de la aplicación se distribuye entre cliente y servidor basándose en la distribución que optimice las configuraciones de cliente y servidor y de la red que los conecta. Por ejemplo, la implementación de una relación mutuamente excluyente implica una búsqueda en la base de datos para determinar si existe un registro que satisfaga los parámetros de una cierta trama de búsqueda. Si no se encuentra ningún registro, se emplea una trama de búsqueda alternativa. Si la aplicación que controla esta trama de búsqueda está contenida en su totalidad en el servidor, se minimiza el tráfico de red. La primera transmisión de la red desde el cliente hacia el servidor contendría los parámetros tanto para la trama de búsqueda primaria como para la trama de búsqueda secundaria. La lógica de aplicación del servidor determinaría si se requiere la segunda búsqueda. El mensaje de repuesta al cliente contendría el registro hallado como consecuencia bien de la primera o bien de la segunda búsqueda. El enfoque alternativo (el cliente implementa la lógica para determinar si se requiere una segunda búsqueda) implicaría un mensaje para la primera recuperación de registros, una respuesta a través de la red **si** no se halla el registro, un segundo mensaje que contuviera los parámetros de la segunda búsqueda, y una respuesta final que contuviera el registro recuperado. Si en la segunda búsqueda se necesita el 50 por 100 de las veces, la colocación de la lógica en el servidor para evaluar la primera búsqueda e iniciar la segunda si fuera necesario reduciría el tráfico de red en un **33** por 100.



Aunque los líneas generoles de lo distribución son valiosos, todos los sistemas deben de tomarse en consideración por sus propios méritos. Poro todos los beneficios derivados de, digamos, un cliente pesado, el diseñador debe de luchar con un conjunto parecido de contruriedodes.

La decisión final acerca de la distribución de componentes debería estar basada no solamente en la aplicación individual, sino en la mezcla de aplicaciones que esté funcionando en el sistema. Por ejemplo, una instalación podría contener algunas aplicaciones que requieran un extenso procesamiento de IGU y muy poco procesamiento central de la base de datos. Esto daría lugar a la utilización de potentes estaciones de trabajo en el lado cliente y a un servidor muy sencillo. Una vez implantada esta configuración, otras aplicaciones favorecerían el enfoque de cliente principal, para que las

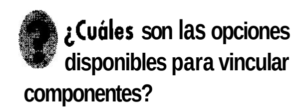
capacidades del servidor no tuvieran necesidad de verse aumentadas.

Habría que tener en cuenta que a medida que madura el uso de la arquitectura cliente/servidor, la tendencia es a ubicar la lógica de la aplicación volátil en el servidor. Esto simplifica la implantación de actualizaciones de software cuando se hacen cambios en la lógica de la aplicación [PAU95].

28.9.4. Enlazado de componentes de software C/S

Se utiliza toda una gama de mecanismos distintos para enlazar los distintos componentes de la arquitectura cliente/servidor. Estos mecanismos están incluidos en la estructura de la red y del sistema operativo, y resultan transparentes para el usuario final situado en el centro cliente. Los tipos más comunes de mecanismos de enlazado son:

- Tuberías *(pipes):* se utilizan mucho en los sistemas basados en UNIX; las tuberías permiten la mensajería entre distintas máquinas que funcionen con distintos sistemas operativos.
- *Llamadas a procedimientos remotos:* permiten que un proceso invoque la ejecución de otro proceso o módulo que resida en una máquina distinta.


¿Cuáles son las opciones disponibles para vincular componentes?

- *Interacción SQL clientelservidor:* se utiliza para pasar solicitudes **SQL** y datos asociados de un componente (típicamente situado en el cliente) a otro componente (típicamente el SGBD del servidor). Este mecanismo está limitado únicamente a las aplicaciones SGBDR.
- *Conexiones (sockets),* se tratan en la Sección 28.6.

Además, las implementaciones orientadas a objetos de componentes de software C/S dan lugar a una &-culación» que haga uso de un distribuidor de solicitudes de objetos. Este enfoque se describirá en la sección siguiente.

28.9.5. Software intermedio (middleware) y arquitecturas de agente de solicitud de objetos

Los componentes de software C/S descritos en las secciones anteriores están implementadas mediante objetos que deben de ser capaces de interactuar entre **sí** en el seno de una sola máquina (bien sea cliente o servidor) o a través de la red. Un agente de distribución de objetos (ORB) es un componente de software intermedio que capacita a un objeto que resida en un cliente para enviar un mensaje a un método que esté encapsulado en otro objeto que resida en un servidor. En esencia, el ORB intercepta el mensaje y maneja todas las actividades de comunicación y de coordinación necesarias para hallar el objeto al cual se había destinado el mensaje, para invocar su método, para pasar al objeto los datos adecuados, y para transferir los datos resultantes de vuelta al objeto que generase el mensaje inicialmente.



Un ORB capacito a un objeto que resida en un cliente para enviar un mensoje a un método que esté encapsulodo en otro objeto que resido en un servidor.

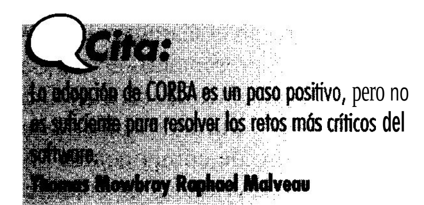
En el Capítulo **27** se estudiaron los tres estándares más utilizados que implementan una filosofía de redistribución de objetos: CORBA, COM y JavaBeans. CORBA se utilizará para ilustrar la utilización del software intermedio ORB.



La información más actualizada sobre estándares de componentes se puede encontrar en la página www.omg.com, **www.microsoft.com/COM,** y **java.sun.com/beans**

En la Figura 28.6 se muestra la estructura básica de una arquitectura CORBA. Cuando se implementa CORBA en un sistema cliente/servidor, los objetos y las clases de objetos (Capítulo 20) tanto en el cliente como en el servidor se definen utilizando un lenguaje de descripción de interfaces (LDI[2]), un lenguaje declarativo que permite que el ingeniero del software defina objetos, atributos, métodos y los mensajes que se requieren para invocarlos. Con objeto de admitir una solicitud para un método residente en el servidor procedente de un objeto residente en el cliente, se crean ***stubs*** LDI en el cliente y en el servidor. Estos *stubs* proporcionan la pasarela a través de la cual se admiten las solicitudes de objetos a través del sistema C/S.

Dado que las solicitudes de objetos a través de la red se producen en el momento de la ejecución, es preciso establecer un mecanismo para almacenar una descripción del objeto de tal modo que la información pertinente acerca del objeto y de su ubicación esté disponible cuando sea necesario. El repositorio de interfaz hace precisamente esto.



Cuando una aplicación cliente necesita invocar un método contenido en el seno de un objeto residente en alguna otra parte del sistema, CORBA utiliza una invo-

[2] En inglés IDL (Interface Description Language)

cación dinámica para (1) obtener la información pertinente acerca del objeto deseado a partir del depósito de interfaz; (2) crear una estructura de datos con parámetros que habrá que pasar al objeto; (3) crear una solicitud para el objeto; y (4) invocar la solicitud. A continuación, se pasa la solicitud al *núcleo ORB* —una parte dependiente de implementación del sistema operativo en red que gestione las solicitudes— y, a continuación, se satisface la solicitud.

La solicitud se pasa a través del núcleo y es procesada por el servidor. En la ubicación del servidor, un *adaptador de objetos* almacena la información de clases y de objetos en un depósito de interfaz residente en el servidor, admite y gestiona las solicitudes procedentes del cliente, y lleva a cabo otras muchas tareas de gestión de objetos [ORF99]. En el servidor unos *stubs* LDI, similares a los definidos en la máquina cliente, se emplean como interfaz con la implementación del objeto real que reside en la ubicación del servidor.

El desarrollo del software para sistemas C/S modernos está orientado a objetos. Empleando la arquitectura CORBA descrita brevemente en esta sección, los desarrolladores de software pueden crear un entorno en el cual se pueden reutilizar los objetos a lo largo y ancho de una red muy amplia. Para más información acerca de CORBA y de su impacto general sobre la ingeniería del software para sistemas C/S, el lector interesado puede consultar [HOQ99] y [SIE99].

28.10 INGENIERÍA DEL SOFTWARE PARA SISTEMAS C/S



En el Capítulo 2 se presentó un cierto número de modelos de proceso de software distintos. Aun cuando muchos de ellos se podrían adaptar para su utilización en el desarrollo de software para sistemas C/S, hay dos enfoques que son los que se utilizan más comúnmente: (1) un paradigma evolutivo que hace uso de la ingeniería del software basada en sucesos y/o orientada a objetos; y (2) una ingeniería del software basada en componentes (Capítulo 27) que se basa en una biblioteca de componentes de software CYD y de desarrollo propio.

Los sistemas cliente servidor se desarrollan empleando las actividades de ingeniería del software clásicas —el análisis, diseño, construcción y depuración— a medida que evoluciona el sistema a partir de un conjunto de requisitos de negocios generales para llegar a ser una colección de componentes de software ya validados que han sido implementados en máquinas cliente y servidor.

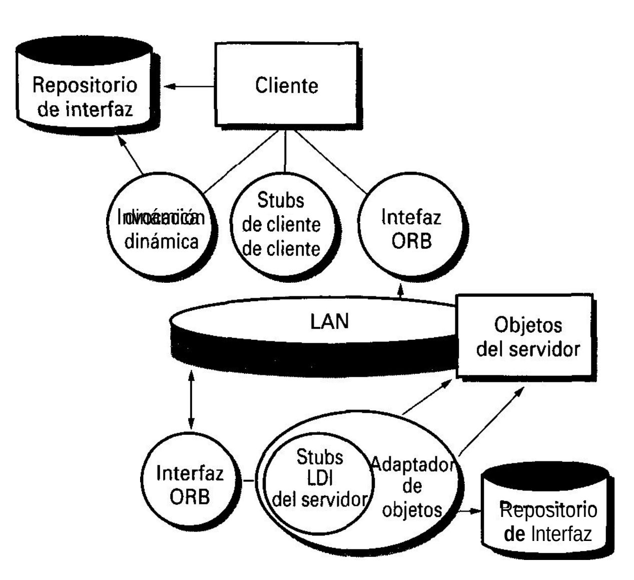


FIGURA 28.6. La arquitectura básica **CORBA.**

28.11 PROBLEMAS DE MODELADO DE ANÁLISIS



La actividad de modelado de requisitos para los sistemas C/S difiere poco de los métodos de modelado de análisis que se aplicaban para la arquitectura de computadoras más convencionales. Consiguientemente, los principios básicos de análisis descritos en el Capítulo 11 y los métodos de modelado de análisis presentados en los Capítulos 12 y 21 son igualmente aplicables al software C/S. Se debería destacar, sin embargo, que dado que muchos sistemas C/S modernos hacen uso de componentes reutilizables, también se aplican las actividades de cualificación asociadas a la ISBC (Capítulo 27).

Dado que el modelado de análisis evita la especificación de detalles de implementación, sólo cuando se haga la transición al diseño se considerarán los problemas asociados a la asignación de software al cliente y al servidor[3]. Sin embargo, dado que se aplica un enfoque evolutivo de la ingeniería del software para los sistemas C/S, las decisiones de implementación acerca del enfoque C/S global (por ejemplo, cliente pesado frente a servidor pesado) se podrán hacer durante las primeras iteraciones de análisis y diseño.

[3] Por ejemplo, una arquitectura C/S conforme a CORBA (Sección 28.1.5) tendrá un profundo impacto en la decisiones sobre el diseño y la implementación.

28.12 DISEÑO DE SISTEMAS C/S

Cuando se está desarrollando un software para su implementación empleando una arquitectura de computadoras concreta, el enfoque de diseño debe de considerar el entorno específico de construcción. En esencia, el diseño debería de personalizarse para adecuarlo a la arquitectura del hardware.

Cuando se diseña software para su implementación empleando una arquitectura cliente/servidor, el enfoque de diseño debe de ser «personalizado» para adecuarlo a los problemas siguientes:

- El diseño de datos (Capítulo 14) domina el proceso de diseño. Para utilizar efectivamente las capacidades de un sistema de gestión de bases de datos relacional (SGBDR) o un sistema de gestión de bases de datos orientado a objetos (SGBDOO) el diseño de los datos pasa a ser todavía más significativo que en las aplicaciones convencionales.



Aun cuando el software C/S es diferente, se pueden utilizar métodos convencionales o de diseño OO con muy pocos modificaciones.

- Cuando se selecciona el paradigma controlado por sucesos, debería realizarse el modelado del comportamiento (una actividad del análisis, Capítulo 12 y 21) y será preciso traducir los aspectos orientados al control implícitos en el modelo de comportamiento al modelo de diseño.
- El componente de interacción/presentación del usuario de un sistema C/S implementa todas aquellas funciones que se asocian típicamente con una interfaz gráfica de usuario (IGU). Consiguientemente, se verá incrementada la importancia del diseño de interfaces (Capítulo 15). El componente de interacción/presentación del usuario implementa todas las funciones que se asocian típicamente con una interfaz gráfica de usuario (IGU).
- Suele seleccionarse un punto de vista orientado a objetos para el diseño (Capítulo 22). En lugar de la estructura secuencia1 que proporciona un lenguaje de procedimientos se proporciona una estructura de objetos mediante la vinculación entre los sucesos iniciados en la IGU y una función de gestión de sucesos que reside en el software basado en el cliente.

Aun cuando prosigue todavía el debate acerca del mejor enfoque de análisis y diseño para los sistemas C/S, los métodos orientados a objetos (Capítulos 21 y 22) parecen ofrecer la mejor combinación de características. Sin embargo, también se pueden adoptar métodos convencionales (Capítulos 12 y 16).

28.12.1. Diseño arquitectónico para sistemas cliente/servidor

El diseño arquitectónico de un sistema cliente servidor se suele caracterizar como un estilo de *comunicación de procesos*. Bass y sus colegas [BAS98] describe esta arquitecturá de la siguiente manera:

> El objetivo es lograr la calidad de la escalabilidad. Un servidor existe para proporcionar datos para uno o más clientes, que suelen estar distribuidos en una red. El cliente origina una llamada al servidor, el cual trabaja síncronamente o asíncronamente, para servir a la solicitud del cliente. Si el servidor funciona síncronamente, devuelve el control al cliente al mismo tiempo que devuelve los datos. Si el servidor trabaja asíncronamente, devuelve solo los datos al cliente (el que tiene su propio hilo de control).

Referencia cruzada

En el Capítulo 14 se presenta un estudio detallado de arquitecturas.

Dado que los sistemas modernos C/S están basados en componentes, se utiliza una arquitectura de agente de solicitud de objetos (ORB) para implementar esta comunicación síncrona y asíncrona.

A un nivel arquitectónico, el lenguaje de descripción de la interfaz CORBA[4] se utiliza para especificar los detalles de la interfaz. La utilización de LDI permite que los componentes de software de la aplicación accedan a los servicios ORB (componentes) sin un conocimiento de su funcionamiento interno. El ORB también tiene la responsabilidad de coordinar la comunicación entre los componentes del cliente y del servidor. Para lograr esto, el diseñador especifica un adaptador de objetos (también llamado *encubridor)* que proporciona los servicios siguientes [BAS98]:

- Se registran las implementaciones de componentes (objetos).
- Se interpretan y se reconcilian todas las referencias de componentes (objetos).
- Se hacen coincidir las referencias de componentes (objetos) con la implementación de los componentes correspondiente.
- Se activan y desactivan objetos.
- Se invocan métodos cuando se transmiten mensajes.
- Se implementan servicios de seguridad.

[4] En COM y JavaBeans se utiliza un método análogo.

Para admitir los componentes CYD proporcionados por proveedores diferentes y componentes de desarrollo propio que pueden haber sido implementados utilizando diferentes tecnologías, se debe diseñar la arquitectura ORB para lograr interoperabilidad entres componentes. Para poderlo llevar a cabo CORBA utiliza un concepto puente.

Supongamos que un cliente se haya implementado utilizando un protocolo ORB X y que el servidor se haya implementado utilizando el protocolo ORB *Y*. Ambos protocolos son conforme CORBA, pero debido a las diferencias de implementación internas, se deben comunicar con un «puente» que proporcione un mecanismo para la traducción entre protocolos internos [BAS98]. El puente traduce mensajes de manera que el cliente y el servidor se puedan comunicar suavemente.

28.12.2. Enfoques de diseño convencionales para software de aplicaciones

En los sistemas cliente/servidor, los diagramas de flujo (Capítulos 12 y 14) se pueden utilizar para establecer el ámbito del sistema, para identificar las funciones de nivel superior y las áreas de datos temáticas (almacenes de datos), y para permitir la descomposición de funciones de nivel superior. Apartándose del enfoque DFD tradicional, sin embargo, la descomposición se detiene en el nivel de un proceso de negocio elemental, en lugar de proseguir hasta el nivel de procesos atómicos.

En el contexto C/S, se puede definir un *proceso elemental de negocios* (PEN) como un cierto conjunto de tareas que se llevan a cabo sin interrupción por parte de un usuario en los centros cliente. Estas tareas o bien se realizan en su totalidad, o bien no se realizan en absoluto.

El diagrama entidad relación adopta también un papel más importante. Sigue utilizándose para descomponer las áreas de datos temáticas (de almacenes de datos) de los DFD con objeto de establecer una visión de alto nivel de la base de datos que haya que implementar empleando un SGBDR. Su nuevo papel consiste en proporcionar la estructura para definir objetos de negocios de alto nivel (Sección 28.4.3).

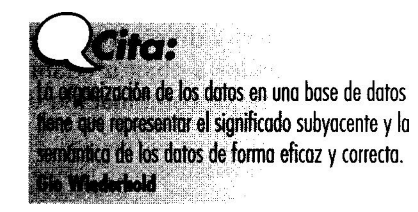
En lugar de servir como herramientas para una descomposición funcional, el diagrama de estructuras se utiliza ahora como diagrama de ensamblaje, con objeto de mostrar los componentes implicados en la solución de algún procedimiento de negocios elemental. Estos componentes, que constan de objetos de interfaz, objetos de aplicación y objetos de bases de datos, establecen la forma en la que se van a procesar los datos.

28.12.3. Diseño de bases de datos

El diseño de bases de datos se utiliza para definir y después para especificar la estructura de los objetos de negocios que se emplean en el sistema cliente/servidor. El análisis necesario para identificar los objetos de negocios se lleva a cabo empleando los métodos de ingeniería de la información descritos en el Capítulo 10. La notación de modelado del análisis convencional (Capítulo 12), tal como DER, se podrá utilizar para definir objetos de negocios, pero es preciso establecer un depósito de base de datos para capturar la información adicional que no se puede documentar por completo empleando una notación gráfica tal como un DER.

En este depósito, se define un *objeto de negocio* como una información que es visible para los compradores y usuarios del sistema, pero no para quienes lo implementan, por ejemplo, un libro en el caso de estudio de ventas de libros. Esta información que se implementa utilizando una base de datos relacional, se puede mantener en un depósito de diseño. La siguiente información de diseño se recoge para la base de datos cliente/servidor [POR94]:

- *Entidades:* se identifican en el DER del nuevo sistema.
- *Archivos:* que implementan las entidades identificadas en el DER.
- *Relación entre campo* y *archivo:* establece la disposición de los archivos al identificar los campos que están incluidos en cada archivo.
- *Campos:* define los campos del diseño (el diccionario de datos).
- *Relaciones entre archivos:* identifican los archivos relacionados que se pueden unir para crear vistas lógicas o consultas.
- *Validación de relaciones:* identifica el tipo de relaciones entre archivos o entre archivos y campos que se utilicen par la validación.
- *Tipos de campo:* se utiliza para permitir la herencia de características de campos procedentes de superenlaces del campo (por ejemplo, fecha, texto, número, valor, precio).
- *Tipo* de *datos:* las características de los datos contenidos en el campo.
- *Tipo de archivo:* se utiliza para identificar cualquiera de las ubicaciones del archivo.
- *Función del campo:* clave, clave externa, atributo, campo virtual, campo derivado, etc.
- *Valorespermitidos:* identifica los valores permitidos para los campos de tipo de estado.
- *Reglas de negocios:* reglas para editar, calcular campos derivados, etc.



A medida que las arquitecturas C/S se han ido haciendo más frecuentes, la tendencia hacia una gestión de datos distribuida se ha visto acelerada. En los sistemas **C/S** que implementan este enfoque, el componente de gestión de datos reside tanto en el cliente como en el servidor. En el contexto del diseño de bases de datos, un problema fundamental es la distribución de datos. Esto es, ¿cómo se distribuyen los datos entre el cliente y el servidor y cómo se dispersan entre los nudos de la red?

Un sistema de gestión de bases de datos relacional (SGBDR) hace fácil el acceso a datos distribuidos mediante el uso del lenguaje de consulta estructurado (SQL). La ventaja de SQL en una estructura C/S es que «no requiere navegar» [BER92]. En un SGBDR, los tipos de datos se especifican empleando SQL, pero no se requiere información de navegación. Por supuesto, la implicación de esto es que SGBDR debe ser suficientemente sofisticado para mantener la ubicación de todos los datos y tiene que ser capaz de definir la mejor ruta hasta ella. En sistemas de bases de datos menos sofisticados, una solicitud de datos debe indicar a qué hay que acceder y dónde se encuentra. Si el software de aplicación debe mantener la información de navegación, entonces la gestión de datos se vuelve mucho más complicada por los sistemas C/S.

¿Qué opciones existen para distribuir datos dentro de un sistema C/S?

Es preciso tener en cuenta que también están disponibles para el diseñador [BER92] otras técnicas para la distribución y gestión de datos:

Extracción manual. Se permite al usuario copiar manualmente los datos adecuados de un servidor a un cliente. Este enfoque resulta útil cuando el usuario requiere unos datos estáticos y cuando se puede dejar el control de la estación en manos del usuario.

Instantánea. Esta técnica automatiza la extracción manual al especificar una «instantánea» de los datos que deberán de transferirse desde un cliente hasta un servidor a intervalos predefinidos. Este enfoque es Útil para distribuir unos datos relativamente estáticos que solamente requieran actualizaciones infrecuentes.

Duplicación. Se puede utilizar esta técnica cuando es preciso mantener múltiples copias de los datos en distintos lugares (por ejemplo, servidores distintos o clientes y servidores). En este caso, el nivel de complejidad se incrementa porque la consistencia de los datos, las actualizaciones, la seguridad y el procesamiento deben de coordinarse entre los múltiples lugares.

Fragmentación. En este enfoque, la base de datos del sistema se fragmenta entre múltiples máquinas. Aunque resulta intrigante en teoría, la fragmentación es excepcionalmente difícil de implementar y hasta el momento no es frecuente encontrarla.

El diseño de bases de datos, y más específicamente, el diseño de bases de datos para sistemas C/S son temas que van más allá del alcance de este libro. El lector interesado puede consultar [BRO91], [BER92], [VAS93] y [ORF99] para una descripción más extensa.

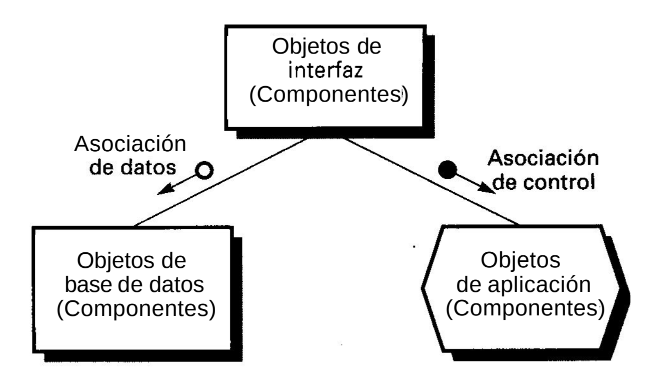


FIGURA 28.7. Notación de diagrama de estructura para componentes C/S.

28.12.4. Visión general de un enfoque de diseño

Porter [POR95] sugiere un conjunto de pasos para diseñar un proceso elemental de negocio que combine elementos de diseño convencional con elementos de diseño orientado a objetos. Se supone que se ha desarrollado un modelo de requisitos que defina los objetos de negocio, y que se ha refinado ya antes de comenzar el diseño de los procesos elementales de negocio. Entonces, se utilizan los pasos siguientes para derivar el diseño:

1. Para cada proceso elemental de negocio, se identifican los archivos creados, actualizados, borrados o referenciados
2. Se utilizan los archivos identificados en el paso 1 como base para definir componentes u objetos.
3. Para cada componente, se recuperan las reglas de negocio y otra información de objetos de negocio que se haya determinado para el archivo relevante.
4. Se determinan las reglas que son relevantes para el proceso y se descomponen las reglas hasta llegar al nivel de métodos.
5. Según sea necesario, se definen los componentes adicionales que se requieren para implementar los métodos.

Porter [POR95] sugiere una notación especializada de diagramas de estructura (Fig. 28.7) para representar la estructura de componentes de un proceso elemental de negocio. Sin embargo, se utiliza una simbología diferente para que el diagrama se ajuste a la naturaleza orientada a objetos del software C/S. En la figura, se encuentran cinco símbolos distintos:

Objeto de interfaz. Este tipo de componente, que también se denomina *componente de interacción/presentación con el usuario*, se construye típicamente en un único archivo o bien otros archivos relacionados que se hayan unido mediante una consulta. Incluye méto-

dos para dar formato a la interfaz IGU y también la lógica de aplicación residente en el cliente, junto con los controles de la interfaz. También incluye sentencias incrustadas de SQL, que especifican el procesamiento de la base de datos efectuado en el archivo primario con respecto al cual se haya construido la interfaz. Si la lógica de aplicación asociada normalmente a un objeto de interfaz se implementa en un servidor, típicamente mediante el uso de herramientas de software intermedio, entonces la lógica de aplicación que funcione en el servidor deberá de identificarse como un objeto de aplicación por separado.

Objeto de base de datos. Este tipo de componentes se utiliza para identificar el procesamiento de bases de datos tal como la creación o selección de registros que esté basada en un archivo distinto del archivo primario en el cual se haya construido el objeto de interfaz. Es preciso tener en cuenta que si el archivo primario con respecto al cual se construye el objeto de interfaz se procesa de manera distinta, entonces se puede utilizar un segundo conjunto de sentencias SQL para recuperar un archivo en una secuencia alternativa. La técnica de procesamiento del segundo archivo debería identificarse por separado en el diagrama de estructura, en forma de un objeto de base de datos por separado.

Objeto de aplicación. Es utilizado por un objeto de interfaz, bien por un objeto de base de datos, y este componente será invocado bien por un activador de una base de datos, o por una llamada a procedimientos remotos. También se puede emplear para identificar la lógica de negocios que normalmente se asocia al procesamiento de interfaz que ha sido trasladado al servidor para su funcionamiento.

Asociación de datos. Cuando un objeto invoca a otro objeto independiente, se pasa un mensaje entre dos objetos. El símbolo de asociación de datos se utiliza para denotar este hecho.

Asociación de control. Cuando un objeto invoca a otro objeto independiente y no se pasan entre los dos objetos, se utiliza un símbolo de asociación de control.

28.12.5. Iteración del diseño de procesos

El repositorio de diseño (Sección 28.4.3), que se utiliza para representar objetos de negocio, se emplea también para representar objetos de interfaz, de aplicación y de base de datos. Obsérvese que se han identificado las entidades siguientes:

- *Métodos:* describen cómo hay que implementar una regla de negocio.
- *Procesos elementales:* definen los procesos elementales de negocio identificados en el modelo de análisis.
- *Enlace procesolcomponente:* identifica a los componentes que forman la solución de un proceso elemental de negocio.
- *Componentes:* describe los componentes mostrados en el diagrama de estructura.
- *Enlace regla de negocioslcornponente:* identifican a los componentes que son significativos para la implementación de una determinada regla de negocio.

Si se implementa un repositorio utilizando un SGBDR, el diseñador tendrá acceso a una herramienta de diseño muy Útil que proporciona informes que sirven de ayuda tanto para la construcción como para el futuro mantenimiento del sistema C/S.

28.13 PROBLEMAS DE LAS PRUEBAS[5]

La naturaleza distribuida de los sistemas cliente/servidor plantea un conjunto de problemas específicos para los probadores de software. Binder [BIN92] sugiere las siguientes zonas de interés:

- Consideraciones del IGU de cliente.
- Consideraciones del entorno destino y de la diversidad de plataformas.
- Consideraciones de bases de datos distribuidas (incluyendo datos duplicados).
- Consideraciones de procesamiento distribuido (incluyendo procesos duplicados).
- Entornos destino que no son robustos.
- Relaciones de rendimiento no lineales.



Se presentan recursos e información útil sobre comprobaciones C/S en la dirección **www.icon-stl.net/-djmosley/**

La estrategia y las tácticas asociadas a la comprobación C/S deben diseñarse de tal modo que se permita abordar todos y cada uno de los problemas anteriores.

28.13.1. Estrategia general de pruebas C/S

En general, las pruebas de software cliente/servidor se produce en tres niveles diferentes: (1) las aplicaciones

[5] Esta sección es una versión muy abreviada y adaptada de un artículo escrito por Daniel Mosley [MOS96] (utilizado con permiso del autor). En [MOS99] se puede encontrar una versión actualizada y ampliada.

de cliente individuales se prueban de modo «desconectado» (el funcionamiento del servidor y de la red subyacente no se consideran); (2) las aplicaciones de software de cliente y del servidor asociado se prueban al unísono, pero no se ejercitan específicamente las operaciones de red; *(3)* se prueba la arquitectura completa de C/S, incluyendo el rendimiento y funcionamiento de la red.

Aun cuando se efectúen muchas clases distintas de pruebas en cada uno de los niveles de detalle anteriores, es frecuente encontrar los siguientes enfoques de pruebas para aplicaciones C/S:

Pruebas defunción de aplicación. Se prueba la funcionalidad de las aplicaciones cliente utilizando los métodos descritos en el Capítulo 17. En esencia, la aplicación se prueba en solitario en un intento de descubrir errores de su funcionamiento.

¿Qué tipos de comprobaciones se realizan para los sistemas C/S?

Pruebas de servidor. Se prueban la coordinación y las funciones de gestión de datos del servidor. También se considera el rendimiento del servidor (tiempo de respuesta y trasvase de datos en general).

Pruebas de bases de datos. Se prueba la precisión e integridad de los datos almacenados en el servidor. Se examinan las transacciones enviadas por las aplicaciones cliente para asegurar que los datos se almacenen, actualicen y recuperen adecuadamente. También se prueba el archivado.

Pruebas de transacciones. Se crea una serie de pruebas adecuada para comprobar que todas las clases de transacciones se procesen de acuerdo con los requisitos. Las transacciones hacen especial hincapié en la corrección de procesamiento y también en los temas de rendimiento (por ejemplo, tiempo de procesamiento de transacciones y comprobación de volúmenes de transacciones).

Pruebas de comunicaciones a través de la red. Estas pruebas verifican que la comunicación entre los nudos de la red se produzca correctamente, y que el paso de mensaje, las transacciones y el tráfico de red relacionado tenga lugar sin errores. También se pueden efectuar pruebas de seguridad de la red como parte de esta actividad de prueba.

Para llevar a cabo estos enfoques de pruebas, Musa [MUS93] recomienda el desarrollo de perfiles operativos derivados de escenarios cliente/servidor. Un *perfil operativo* indica la forma en que los distintos tipos de usuarios interactúan con el sistema C/S. Esto es, los perfiles proporcionan un «patrón de uso» que se puede aplicar cuando se diseñan y ejecutan las pruebas. Por ejemplo, para un determinado tipo de usuarios, ¿qué porcentaje de las transacciones serán consultas? ¿Actualizaciones? ¿Pedidos?

Referencia cruzada

Las técnicas de elicitación de requisitos y los casos prácticos de estudio se tratan en el Capítulo 11.

Para desarrollar el perfil operativo, es preciso derivar un conjunto de escenarios de usuario [BIN95], que sea similar a los casos prácticos de estudio tratados en este libro. Cada escenario abarca quién, dónde, qué y por qué. Esto es, quién es el usuario, dónde (en la arquitectura física) se produce la interacción con el sistema, cuál es la transacción, y por qué ha sucedido. Se pueden derivar los escenarios durante las técnicas de obtención de requisitos o a través de discusiones menos formales con los usuarios finales. Sin embargo, el resultado debería ser el mismo. Cada escenario debe proporcionar una indicación de las funciones del sistema que se necesitarán para dar servicio a un usuario concreto; el orden en que serán necesarias esas funciones, los tiempos y respuestas que se esperan, y la frecuencia con la que se utilizará cada una de estas funciones. Entonces se combinan estos datos (para todos los usuarios) para crear el perfil operativo.

La estrategia para probar arquitecturas C/S es análoga a la estrategia de pruebas para sistemas basados en software que se describían en el Capítulo 18. La comprobación comienza por comprobar al por menor. Esto es, se comprueba una única aplicación de cliente. La integración de los clientes, del servidor y de la red se irá probando progresivamente. Finalmente, se prueba todo el sistema como entidad operativa. La prueba tradicional visualiza la integración de módulos para subsistemas/sistemas y su prueba (Capítulo 18) como un proceso descendente, ascendente o alguna variación de los dos anteriores. La integración de módulos en el desarrollo C/S puede tener algunos aspectos ascendentes o descendentes, pero la integración en proyectos C/S tiende más hacia el desarrollo paralelo y hacia la integración de módulos en todos los niveles de diseño. Por tanto, la prueba de integración en proyectos C/S se efectúa a veces del mejor modo posible empleando una aproximación no incremental o del tipo «*big bang*».

Cita:

El tópico de la comprobación es un área en donde elementos comunes se encuentran entre los sistemas tradicionales y los sistemas basados en cliente/servidor.

Kelley Bourne

El hecho de que el sistema no se esté construyendo para utilizar un hardware y un software especificado con anterioridad tiene su impacto sobre la comprobación del sistema. La naturaleza multiplataforma en red de los sistemas C/S requiere que se preste bastante más atención a la prueba de configuraciones y a la prueba de compatibilidades.

www.elsolucionario.net

La doctrina de prueba de configuraciones impone la prueba del sistema en todos los entornos conocidos de hardware y software en los cuales vaya a funcionar. La prueba de compatibilidad asegura una interfaz funcionalmente consistente entre plataformas de software y hardware. Por ejemplo, la interfaz de ventanas puede ser visualmente distinta dependiendo del entorno de implementación, pero los mismos comportamientos básicos del usuario deberían dar lugar a los mismos resultados independientemente del estándar de interfaz de cliente.



Especificación de pruebas C/S

28.13.2. Táctica de pruebas C/S

Aun cuando el sistema C/S no se haya implementado empleando tecnología orientada a objetos, las técnicas de pruebas orientadas a objetos (Capítulo 23) tienen sentido porque los datos y procesos duplicados se pueden organizar en clases de objetos que compartan un mismo conjunto de propiedades. Una vez que se hayan derivado los casos prácticos para una clase de objetos (o su equivalente en un sistema desarrollado convencionalmente), estos casos de prueba deberían resultar aplicables en general a todas las instancias de esa clase.

El punto de vista OO es especialmente valioso cuando se considera la interfaz gráfica de usuario de los sistemas C/S modernos. La IGU es inherentemente orientada a objetos, y se aparta de las interfaces tradicionales porque tiene que funcionar en muchas plataformas. Además, la comprobación debe explorar un elevado número de rutas lógicas, porque la IGU crea, manipula y modifica una amplia gama de objetos gráficos. La comprobación se ve complicada aun más porque los objetos pueden estar presentes o ausentes, pueden existir durante un cierto período de tiempo, y pueden aparecer en cualquier lugar del escritorio.

Esto significa que el enfoque tradicional de captura/reproducción para comprobar interfaces convencionales basadas en caracteres debe ser modificado para que pueda manejar las complejidades de un entorno IGU. Ha aparecido una variación funcional del paradigma de captura/reproducción denominado *captural reproducción estructurada* [FAR93] para efectuar la prueba de la IGU.

La captura/reproducción tradicional registra las entradas tales como las teclas pulsadas y las salidas tales como imágenes de pantalla que se almacenan para comprobaciones posteriores. La captura/reproducción estructurada está basada en una visión interna (lógica) de las actividades externas. Las interacciones del programa de aplicación con la IGU se registran como sucesos internos, que se pueden almacenar en forma de «guiones» escritos en Visual Basic de Microsoft, en una de las variantes de **C,** o en el lenguaje patentado del fabricante. Las herramientas que ejercitan las IGUs no abarcan las validaciones de datos tradicionales ni tampoco las necesidades de comprobación de rutas. Los métodos de comprobación de caja blanca y caja negra descritos en el Capítulo 17 son aplicables en muchos casos, y las estrategias orientadas a objetos especiales que se presentaban en el Capítulo 23 son adecuadas tanto para el software de cliente como para el software de servidor.

RESUMEN

Aun cuando los sistemas cliente/servidor pueden adoptar uno o más de los modelos de procesos de software y muchos de los métodos de análisis, diseño y comprobación descritos anteriormente en este libro, las características de arquitecturas especiales de C/S requieren una personalización del software de ingeniería del software. En general, el modelo de proceso del software que se aplica a los sistemas C/S tiene una naturaleza evolutiva, y los métodos técnicos suelen tender a enfoques orientados a objetos. El desarrollador debe describir aquellos objetos que se produzcan en la implementación de los componentes de interacción /representación con el usuario, de base de datos y de aplicación. Los objetos definidos para estos componentes deberían asignarse bien a la máquina, o bien a la máquina servidor, y se pueden vincular a través de un distribuidor de solicitudes de objetos.

Las arquitecturas de agente de solicitud de objetos sirven de apoyo para los diseños C/S en los cuales los objetos cliente envían mensajes a los objetos servidor. El estándar CORBA hace uso de un lenguaje de definición de interfaz, y los repositorios de interfaz gestionan las solicitudes de objetos independientemente de su ubicación dentro de la red.

El análisis y el diseño de sistemas cliente/servidor hacen uso de diagramas de flujo de datos **y** de entidades y relaciones, diagramas de estructura modificados, y otras notaciones que se encuentran en el desarrollo de aplicaciones convencionales. **Es** preciso modificar las estrategias de comprobación para adecuarlas a las comprobaciones que examinan la comunicación a través de la red y el juego entre el software que reside en el cliente y aquel que reside en el servidor.

REFERENCIAS

[BAS98] Bass, L., P. Clemens y R. Kazman, ***Software Architecture in Practice,*** Addison-Wesley, 1998.

[BER92] Berson, A., ***ClientlServer Architecture,*** McGraw-Hill, 1992.

[BIN92] Binder, R., «A CASE-based Systems Engineering Approach to Client-Server Development», ***CASE Trends,*** 1992.

[BIN95] Binder, R., «Scenario-Based Testing for Client Server systems», ***Software Development,*** vol. 3, n.º 8, Agosto de 1995, pp. 43-49.

[BRO91] Brown, A.W., ***Object-Oriented Databases,*** McGraw-Hill, 1991.

[FAR93] Farley, K.J., «Software Testing For Windows Developers», ***Data Based Advisor***, Noviembre de 1993, pp. 45-46, 50-52.

[HOQ99] Hoque, R., ***CORBAfor Real Programmers***, Academic Press/Morgan Kaufmann, 1999.

[MOS99] Mosley, D., ***Client Server Software Testing on the Desk Top and the Web,*** Prentice-Hall, 1999.

[MUS93] Musa, J., «Operational Profiles in Software Reliability Engineering», ***IEEE Software,*** Marzo de 1993, pp. 14-32.

[ORF99] Orfali, R., D. Harkey y J. Edwards, ***Essential ClientlServer Survival Guide,*** 3.ª ed., Wiley, 1999.

[PAU95] L. G. Paul, «Client/Server Deployment», ***Computerworld***, 18 de Diciembre, 1995.

[POR94] Porter, J., ***O-DES Design Manual,*** Fairfield University, 1994.

[POR95] Porter, J., ***Synon Developer's Guide,*** McGraw-Hill, 1995.

[REE97] Reece, G., ***JBDC and Java,*** O'Reilly, 1997.

[SIE99] Siegel, J., ***CORBA 3 Fundamentals and Programming,*** Wiley, 1999.

[VAS93] Vaskevitch, D., ***ClientlServer Strategies,*** IDG Books. 1993.

PROBLEMAS Y PUNTOS A CONSIDERAR

28.1. Empleando publicaciones comerciales ***o*** recursos e Internet de información de fondo, defina un conjunto de criterios para evaluar herramientas para la ingeniena del software C/S.

28.2. Sugiera cinco aplicaciones en las cuales un servidor pesado parezca una estrategia de diseño adecuada.

28.3. Sugiera cinco aplicaciones en las cuales un cliente pesado parezca ser una estrategia de diseño adecuada.

28.4. Efectúe una investigación sobre los Últimos delitos cometidos en Intemet y describa cómo la tecnología de seguridad apuntada en este capítulo podría haberla evitado.

28.5. Describa cómo se podría estructurar un sistema de reservas en unas líneas aéreas como un sistema cliente/servidor.

28.6. Investigue los últimos avances en software para trabajo en grupo y desarrolle una breve presentación para la clase. El profesor puede asignar una función específica a los distintos participantes.

28.7. Una compañía va a establecer una nueva división de ventas por catálogo para vender mercancías de uso frecuente y en exteriores. El catálogo electrónico se publicará en la World Wide Web, y se pueden hacer pedidos a través de correo electrónico, mediante el sitio Web y también por teléfono ***o*** fax. Se construirá un sistema cliente/servidor para prestar su apoyo al procesamiento de pedidos en el sitio de la compañía. Defina un conjunto de objetos de alto nivel que fueran necesarios para el sistema de procesamiento de pedidos, y organice estos objetos en tres categorías de componentes: la presentación con el usuario, la base de datos y la aplicación.

28.8. Formule reglas de negocio para establecer cuándo se puede efectuar un envío si el pago se efectúa mediante tarjeta de crédito, con respecto al sistema descrito en el Problema 28.7. Añada reglas adicionales si el pago se efectúa mediante cheque.

28.9. Desarrolle un diagrama de transición de estados (Capítulo 12) que defina los sucesos y estados que resultarían visibles para un dependiente de entrada de pedidos que trabajase en un PC cliente en la división de ventas por catálogo (Problema 28.7).

28.10. Ofrezca ejemplos de tres ***o*** cuatro mensajes que pudieran dar lugar a una solicitud de un método de cliente mantenido en el servidor.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Aun cuando los métodos básicos de análisis y diseño para sistemas cliente/servidor son bastante parecidos a los sistemas OO convencionales, se requieren técnicas y conocimientos especializados. Lowe y Helda han escrito unas introducciones valiosas a los conceptos ***básicos (Client/Server Computingfor Dummies,*** 3.ª ed., IDG Books Worldwide, 1999), al igual que Zantinge y Adriaans ***(ManagingClient/Server,*** Addison-Wesley, 1997). En un nivel intermedio, McClanahan ***(Developing Client-Server Applications,*** IDG Books Worldwide, 1999) abarca un amplio abanico de temas C/S. A un nivel más sofisticado, Orfali y sus colaboradores [ORF99], y Linthicum ***(Guide to ClientlServer and Intranet Development,*** Wiley, 1997) proporcionan las líneas generales detalladas para aplicaciones C/S. Berson ***(ClientlServer Architecture,*** 2.ª ed., McGraw-Hill, 1996) trata temas de arquitectura y componentes.

En los últimos años las computadoras de redes se han convertido en un tema tecnológico candente (y una estrategia arriesgada de negocios). Sinclair y Merkow *(Thin Clients Clearly Explained,* Morgan Kaufmann Publishers, 1999), Friedrichs y Jubin *(Java Thin-Client Programrning for a Network Computing Environment,* Prentice-Hall, 1999), Dewire *(Thin Clients,* McGraw-Hill, 1998) y Kanter *(Understanding Thin-Client/Server Cornputing,* Microsoft Press, 1998) proporcionan una guía valiosa de cómo diseñar, construir, hacer uso y apoyar sistemas cliente ligeros (delgados).

Empezando con el modelado de sucesos de negocios, Ruble *(Practica1 Analysis and Design for ClientlServer and GUI Systems,* 1997) proporciona un estudio en profundidad de las técnicas para el análisis y diseño de sistemas C/S. Los libros de Goldman, Rawles y Mariga *(ClientlServer Information Systems: A Business-Oriented Approuch,* Wiley, 1999), Shan, Earle y Lenzi *(Enterprise Computing With Objects: From ClientlServer Environments to the Internet,* Addison-Wesley, 1997) y Gold-Berstein y Marca *(Designing Enterprise ClientlServer Systems,* Prentice-Hall, 1997) tienen en consideración **los** sistemas C/S en un contexto más amplio de empresa.

Existe un grupo de libros que estudian la seguridad en redes; a continuación se proporciona una muestra:

Stallings, W., *Cryptology and Network Security,* Prentice-Hall, 1998.

Gralla, P., *The Complete Idiots Guide to Protecting Yourself Online,* Que, 1999.

Ghosh, **A.** K., *E-cornrnerce Security: Weak Links, Best Defenses*, John Wiley, 1998.

Atkins, D. T., Shelden y T. Petru, *Internet Security: Professional Reference*, New Riders Publishing, 1997.

Bernstein, T., A.B. Bhimani, E. Schultz y C. Siegel, *Internet Security for Business,* John Wiley, 1998.

Garfinkel, S., y G. Spafford, *Web Security and Commerce, O.* Reilly, 1997.

Tiwana, A., *Web Security,* Digital Press, 1999.

Loosely y Douglas *(High-Performance ClientlServer,* Wiley, 1997) explican **los** principios de la ingeniería de rendimiento del software y **los** aplican a la arquitectura y al diseño de los sistemas distribuidos. Heinckiens **y** Loomies *(Building Scalable Database Applications: Object-Oriented Design, Architectures, and Implementations*, Addison-Wesley, 1998) dan importancia al diseño de bases de datos en **su** guía para construir aplicaciones cliente/servidor. Ligon *(ClientlServer Comrnunicutions Services: A Guide for the Applications Developer,* McGraw-Hill, 1997) tiene en cuenta una gran variedad de temas de comunicación relacionados entre los que se incluyen TCP/IP, ATM, EDI, CORBA, mensajes y encriptación. Schneberger *(ClientlServer Software Maintenance,* McGraw-Hill, 1997) presenta un marco de trabajo para controlar **los** costes de mantenimiento de software C/S y para optimizar el apoyo al usuario.

Literalmente cientos de libros abarcan el desarrollo de sistemas C/S específicos del vendedor. La siguiente relación representa un pequeño muestreo:

Anderson, G.W., Client/Server Database Design With Sybase: A High-Performance and Fine-Tuning Guide, McGraw-Hill, 1997.

Barlotta, M. J., Distributed Application Development With Powerbuilder ***6*,** Manning Publications Company, 1998.

Bates, R.J., Hands-On Client/Server Internetworking, McGraw-Hill, 1997.

Mahmoud, Q.H., Distributed Programming with java, Manning Publications Company, 1998.

Orfali, R., y Harkey, Client/Server Programming with JavaBeans, Wiley, 1999.

Sankar, K., Building Internet Client/Server Systems, McGraw-Hill, 1999.

Mosley [MOS99] y Boume *(Testing ClientlServer Systerns,* McGraw-Hill, 1997) han escrito libros de guía detallados para pruebas C/S. Ambos autores proporcionan un estudio en profundidad de las estrategias de comprobación, tácticas y herramientas.

Una gran variedad de fuentes de información sobre ingeniería del software cliente/servidor está disponible en Internet. Una lista actualizada de referencias a sitios (páginas) web que son relevantes para sistemas C/S se pueden encontrar en **http://www.pressman5.com.**

CAPÍTULO

29 INGENIERÍA WEB

LA World Wide Web e Internet han introducido a la población en general en el mundo de la informática. Compramos fondos de inversión colectivos y acciones, descargamos música, vemos películas, obtenemos asesoramiento médico, hacemos reservas de habitaciones en hoteles, vendemos artículos personales, planificamos vuelos en líneas aéreas, conocemos gente, hacemos gestiones bancarias, recibimos cursos universitarios, hacemos la compra **-es** decir, en el mundo virtual se puede hacer todo lo que se necesite**—**. Se puede decir que Internet y la Web son los avances más importantes en la historia de la informática. Estas tecnologías informáticas nos han llevado a todos nosotros a la era de la informática (con otros millones de personas quienes finalmente entrarán también). Durante los primeros años del siglo veintiuno estas tecnologías han llegado casi a formar parte de nuestra vida diaria.

Para todos nosotros que recordamos un mundo sin Web, el crecimiento caótico de la tecnología tiene su origen en otra era —los primeros días del software**—**. Eran tiempos de poca disciplina, pero de enorme entusiasmo y creatividad. Eran tiempos en que los programadores a menudo entraban en otros sistemas, algunas veces con buena intención y otras con mala intención. La actitud que prevalecía parecía ser la de «Hazlo rápidamente, y entra en el campo, que nosotros lo limpiaremos **(y** mejor sería que entendieras lo que realmente queremos construir) cuando actuemos». ¿Le suena familiar?

En una mesa redonda virtual publicada en IEEE Software [PRE98], mantuve en firme mi postura en relación con la ingeniería de Web:

> Me parece que cualquier producto o sistema importante es merecedor de recibir una ingeniería. Antes de comenzar a construirlas, lo mejor es entender el problema, diseñar una solución viable, implementarla de una manera sólida y comprobarla en profundidad. Probablemente también se deberían controlar **los** cambios a medida que el trabajo vaya avanzando, y disponer de mecanismos para asegurar la calidad del resultado final. Muchos de los que desarrollan Webs no dicen lo mismo; ellos piensan que su mundo es realmente diferente, y que simplemente no se van a aplicar los enfoques de ingeniería del software convencionales.

VISTAZO RÁPIDO

¿Qué es? Los sistemas y aplicaciones (WebApps) basados en Web hacen posible que una población extensa de usuarios finales dispongan de una gran variedad de contenido y funcionalidad. La ingeniería Web no es un clónico perfecto de la ingeniería del software, pero toma prestado muchos de los conceptos y principios básicos de la ingeniería del software, dando importancia a las mismas actividades técnicas y de gestión. Existen diferencias sutiles en la forma en que se llevan a cabo estas actividades, pero la filosofía primordial es idéntica dado que dicta un enfoque disciplinado para el desarrollo de un sistema basado en computadora.

¿Quién lo hace? Los ingenieros Web y los desarrolladores de contenido no técnicos crean las WebApps.

¿Por qué es importante? A medida que las WebApps se integran cada vez más en grandes y pequeñas compañías (por ejemplo, comercio electrónico), y cada vez es más importante la necesidad de construir sistemas fiables, utilizables y adaptables. Esta es la razón por la que es necesario un enfoque disciplinado para el desarrollo de WebApps.

¿Cuáles son los pasos a *seguir*? Al igual que cualquier disciplina de ingeniería, la ingeniería Web aplica. un enfoque genérico que se suaviza con estrategias, tácticas y métodos especializados. El proceso de ingeniería Web comienza con una formulación del problema que pasa a resolverse con las WebApps. Se planifica el proyecto y se analizan los requisitos de la WebApp, entonces se lleva a cabo el diseño de interfaces arquitectónico y del navegador. El sistema se implementa utilizando lenguajes **y** herramientas especializados asociados con la Web, y entonces comienzan las pruebas. Dado que las WebApps están en constante evolución, deben de establecerse los mecanismos para el control de configuraciones, garantía de calidad y soporte continuado.

¿Cuál es el producto obtenido? La elaboración de una gran variedad de productos de trabajo de ingeniería Web (por ejemplo, modelos de análisis. modelos de diseño, procedimientos de pruebas). **Y** como producto final la WebApp operativa.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Aplicando las mismas prácticas SQA que *se* aplican en todos los procesos de ingeniería del software —las revisiones *técnicas* formales valoran los modelos de análisis y diseño—; las revisiones especializadas tienen en consideración la usabilidad y la comprobación se aplica para descubrir errores en el contenido, funcionalidad y compatibilidad.

Cita:
Los principios de planificación previos al diseño de la ingeniería y el diseño previo a la construcción, se han alimentado de todas las transiciones anteriores de la tecnología; por tanto, también sobrevivirán a esta transición.
Watts Humphrey

Esto nos lleva a formular una pregunta clave: ¿Pueden aplicarse principios, conceptos y métodos de ingeniería en el desarrollo de la Web? Creo que muchos de ellos **sí** se pueden aplicar, pero su aplicación quizás requiera un giro algo diferente.

Pero, ¿qué ocurriría si estuviera equivocado?, y ¿si persiste el enfoque actual y específico para ese desarrollo de la Web? Con la ausencia de un proceso disciplinado para sistemas basados en Web, cada vez nos preocupa más la manera en que nos podemos enfrentar con problemas serios para obtener éxito en el desarrollo, empleo y «mantenimiento» de estos sistemas. En esencia, a medida que entramos en el nuevo siglo, la infraestructura de las aplicaciones que se están creando hoy en día puede llevamos a algo que se podría llamar «Web enmarañada». Esta frase connota un cúmulo de aplicaciones basadas en Web pobremente desarrolladas y con una probabilidad de fallo bastante alta. Y lo que es peor, a medida que los sistemas basados en Web se van complicando, un fallo en uno de ellos puede propagar y propagará problemas muy extensos en todos. Cuando ocurra esto, la confianza en Internet se puede romper provocando resultados irremediables. Y lo que es aún peor, puede conducir a una regulación gubernamental innecesaria y mal concebida, provocando daños irreparables en estas tecnologías singulares.

Con objeto de evitar una Web enmarañada y lograr un mayor éxito en el desarrollo y aplicación de sistemas basados en Web complejos y a gran escala, existe una necesidad apremiante de enfoques de ingeniería Web disciplinada y de métodos y herramientas nuevos para el desarrollo, empleo y evaluación de sistemas y aplicaciones basados en Web. Tales enfoques y técnicas deberán tener en cuenta las características especiales en el medio nuevo, en los entornas y escenarios operativos, y en la multiplicidad de perfiles de usuario implicando todo ello un reto adicional para el desarrollo de aplicaciones basadas en Web.

La Ingeniería Web (IWeb) está relacionada con el establecimiento y utilización de principios científicos, de ingeniería y de gestión, y con enfoques sistemáticos y disciplinados del éxito del desarrollo, empleo y mantenimiento de sistemas y aplicaciones basados en Web de alta calidad [MUR99].

29.1 LOS ATRIBUTOS DE APLICACIONES BASADAS EN WEB

No hay mucho que decir con respecto al hecho de que los sistemas y las aplicaciones' basados en Web (nos referiremos a estas como *WebApps*) son muy diferentes de las otras categorías de software informático que se tratan en el Capítulo 1. Powell resume las diferencias básicas cuando afirma que los sistemas basados en Web «implican una mezcla de publicación impresa y desarrollo de software, de marketing e informática, de comunicaciones internas y relaciones externas, y de arte y tecnología» [POW98]. Los atributos siguientes se van a encontrar en la gran mayoría de las WebApps[2]:

Intensivas de Red. Por su propia naturaleza, una WebApp es intensiva de red. Reside en una red y debe dar servicio a las necesidades de una comunidad diversa de clientes. Una WebApp puede residir en Internet (haciendo posible así una comunicación abierta par todo el mundo). De forma alternativa, una aplicación se puede ubicar en una Intranet (implementando la comunicación a través de redes de una organización) o una Extranet (comunicación entre redes).



las WebApps son intensivas de red, controladas por el contenido y en continua evolución. Estos atributos tienen un profundo impacto dentro de la forma en que se lleva a coba la IWeb.

Controlada por el contenido. En muchos casos, la función primaria de una WebApp es utilizar hipermedia para presentar al usuario el contenido de textos, gráficos, sonido y vídeo.

[1] En esta categoria se incluyen unos sitios Web completos, funcio nalidad especializada dentro de los sitios Web y aplicaciones de proceso de informacion que residen en Internet, en una Intranet o en una Extranet

[2] En el contexto de este capitulo el termino «aplicación Web» abarcara todo, desde una pagina Web simple que podria ayudar a un consumidor a calcular el pago de un alquiler de un coche hasta un sitio Web completo que proporcione los servicios completos para viales de negocios y de vacaciones

Evolución continua. A diferencia del software de aplicaciones convencional, que evoluciona con una serie de versiones planificadas y cronológicamente espaciadas, las aplicaciones Web están en constante evolución. No es inusual que algunas WebApps (específicamente, su contenido) se actualicen cada hora.

Algunos argumentan que la evolución continua de las WebApps hace que el trabajo realizado en ellas sea similar a la jardinería. Lowe [LOW99] trata este tema con el siguiente escrito:

> **La ingeniena está a punto de adoptar un enfoque científico y consecuente, suavizado por un contexto específico y práctico, para el desarrollo y el comisionado de sistemas y aplicaciones. El desarrollo de los sitios Web suele estar destinado a crear una infraestructura (sembrar el jardín) y entonces «ocuparse» de la información que crece y brota dentro del jardín. Después de un tiempo, el jardín (es decir, el sitio Web) continuará evolucionando, cambiando y creciendo. Una buena arquitectura inicial deberá permitir que este crecimiento ocurra de forma controlada y consecuente...** podríamos hacer que «tres cirujanos» **podaran los «árboles» (es decir, las secciones del sitio Web) dentro del jardín (el sitio en sí) a la vez se asegura la integridad de las referencias cruzadas. Podríamos disfrutar de «guarderías de** jardín» **donde** «se cultiven» **las plantas jóvenes (es decir, las configuraciones de diseño para sitios Web). Acabemos con esta analogía, no vaya a ser que vayamos demasiado lejos.**

Un cuidado y una alimentación continua permite que un sitio Web crezca (en robustez y en importancia). Pero a diferencia de un jardín, las aplicaciones Web deben de servir (y adaptarse a) las necesidades de más de un jardinero, Las siguientes características de WebApps son las que conducen el proceso:

Inmediatez. Las aplicaciones basadas en Web tienen una inmediatez [NOR99] que no se encuentra en otros tipos de software. **Es** decir, el tiempo que se tarda en comercializar un sitio Web completo puede ser cuestión de días o semanas[3]. Los desarrolladores deberán utilizar los métodos de planificación, análisis, diseño, implementación y comprobación que se hayan adaptado a planificaciones apretadas en tiempo para el desarrollo de WebApps.



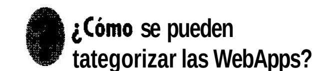
No hay duda de que la inmediatez a menudo prevalece en el desarrollo de las WebApps, pero hay que tener cuidado, porque el hecho de hacer algo rápidamente no significo tener el privilegio de hacer un trabajo de ingenieríopobre. lo rápido y erróneo raras veces tiene un resultado acepiable.

Seguridad. Dado que las WebApps están disponibles a través del acceso por red, es difícil, si no imposible, limitar la población de usuarios finales que pueden acceder a la aplicación. Con objeto de proteger el contenido confidencial y de proporcionar formas seguras de transmisión de datos, deberán implementarse fuertes medidas de seguridad en toda la infraestructura que apoya una WebApp y dentro de la misma aplicación.

Estética. Una parte innegable del atractivo de una WebApp es su apariencia e interacción. Cuando se ha diseñado una aplicación con el fin de comercializarse o vender productos o ideas, la estética puede tener mucho que ver con el éxito del diseño técnico.

Las características generales destacadas anteriormente se aplican a todas las WebApps, pero con un grado diferente de influencia. Las categorías de aplicaciones que se enumeran a continuación son las más frecuentes en el trabajo de la Web [DAR99]:

- *informativa:* se proporciona un contenido solo de lectura con navegación y enlaces simples;
- *descarga:* un usuario descarga la información desde el servidor apropiado;



- *personalizable:* el usuario personaliza el contenido a sus necesidades específicas;
- *interacción:* la comunicación entre una comunidad de usuarios ocurre mediante un espacio *chat* (charla), tablones de anuncios o mensajería instantánea;
- *entrada del usuario:* la entrada basada en formularios es el mecanismo primario de la necesidad de comunicación;
- *orientada a transacciones:* el usuario hace una solicitud (por ejemplo, la realización un pedido) que es cumplimentado por la WebApp;
- *orientado a servicios:* la aplicación proporciona un servicio al usuario, por ejemplo, ayuda al usuario a determinar un pago de hipoteca;
- *portal:* la aplicación canaliza al usuario llevándolo a otros contenidos o servicios Web fuera del dominio de la aplicación del portal;
- *acceso a bases de datos:* el usuario consulta en una base de datos grande y extrae información;
- *almacenes de datos:* el usuario hace una consulta en una colección de bases de datos grande y extrae información.

Las características y las categorías destacadas anteriormente en esta sección, y las categorías de aplicaciones representan los hechos reales para los ingenieros de la Web. La clave es vivir dentro de las restricciones impuestas por las características anteriores y aun así tener éxito en la elaboración de la WebApp.

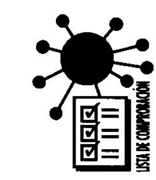
29.1.1. Atributos de calidad

Todas las personas que hayan navegado alguna vez por la Web o hayan utilizado una intranet de una compañía pue-

[3] **Páginas Web** sofisticadas **pueden elaborarse en pocas horas**

den opinar sobre lo que hace una «buena» WebApp. Los puntos de vista individuales varían enormemente. Algunos usuarios disfrutan con gráficos llamativos, en cambio otros solo quieren un texto sencillo. Algunos exigen información copiosa, otros desean una presentación abreviada. En efecto, la percepción de «lo bueno» por parte del usuario (y como consecuencia, la aceptación o no aceptación resultante de la WebApp) podría ser más importante que cualquier discusión técnica sobre la calidad de la WebApp.

Pero ¿cómo se percibe la calidad de la WebApp? ¿qué atributos deben de exhibirse ante los ojos de los usuarios para lograr lo bueno y al mismo tiempo exhibir las características técnicas de calidad que permitan a un ingeniero corregir, adaptar, mejorar y soportar la aplicación a largo plazo?



Arbol detallado de los requisitos de calidad para WebApps

En realidad, todas las características generales de la calidad del software estudiadas en los Capítulos **8,** 19 y 24 se aplican también a las WebApps. Sin embargo, las características más relevantes —usabilidad, fiabilidad, eficiencia y capacidad de mantenimiento— proporcionan una base útil para evaluar la calidad de los sistemas basados en Web.

Olsina y sus colaboradores [OSL99] han preparado un «árbol de requisitos de calidad» que identifica un conjunto de atributos que conduce a WebApps de alta calidad. La Figura 29.1 resume su trabajo.

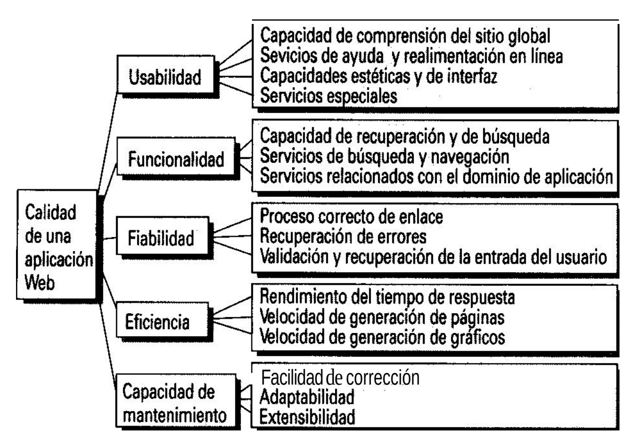


FIGURA 29.1. Árbol de requisitos de calidad (OSL 99).

29.1.2. Las tecnologías

El diseño y la implementación de sistemas basados en Web incorporan tres tecnologías importantes: el desarrollo basado en componentes, la seguridad y los estándares de Internet. Un ingeniero Web deberá estar familiarizado con las tres para construir WebApps de alta calidad.

Desarrollo basado en componentes

Las tecnologías de componentes estudiadas en los Capítulos 27 y 28 han evolucionado en gran parte gracias al crecimiento explosivo de los sistemas y aplicaciones basados en Web. Retomando el estudio del capítulo anterior, los ingenieros Web disponen de tres estándares importantes para la infraestructura: CORBA, COM/DCOM y JavaBeans. Estos estándares (acompañados por los componentes preconstmidos, herramientas y técnicas) proporcionan una infraestructura que permite a los que diseñan emplear y personalizar componentes de terceras partes permitiéndoles así comunicarse unos con otros y con servicios a nivel de sistemas.

Cita:
Internet es un lugar arriesgado para dirigir un negocio o para controlar las mercancías de un almacén. Por todos lados nos podemos encontrar intrusos informáticos (hackers), destructores de sistemas informáticos (crackers), fisgones, chanchulleros, impostores, metomentodos, intrusos, ladrones, hurtadores, conspiradores, vándalos, vendedores de Caballo de Troya, emisores de virus y proveedores de programas malévolos.
Dorothy Deningy Peter Denning

Seguridad

Si en una red reside una WebApp, ésta está abierta a un acceso sin autorización. En algunos casos, ha sido el personal interno el que ha intentado acceder sin autorización. En otros casos, intrusos (*hackers*) pueden intentar acceder por deporte, por sacar provecho o con intenciones más maliciosas. Mediante la infraestructura de red se proporciona una variedad de medidas de seguridad, tales como encriptación, cortafuegos y otras. Un estudio amplio de este tema queda fuera del ámbito de este libro. Para más información, el lector interesado puede consultar en esta bibliografía: [ATK97], [KAE99] y [BRE99].

Estándares de Internet

Durante la última década el estándar dominante en la creación del contenido y la estructura de la WebApp ha sido HTML, un lenguaje de marcas que posibilita al desarrollador proporcionar una serie de etiquetas que describen una gran variedad de objetos de datos (texto, gráficos, audio/vídeo, formularios, etc.). Sin embargo, a medida que las aplicaciones crecen en tamaño y complejidad, se ha adoptado un nuevo estándar —XML— para la próxima generación de WebApps. XML (Extensible Markup Language) el Lenguaje de marcas extensible es un subconjunto estrictamente definido del metalenguaje SGML [BRA97], permitiendo que los diseñadores definan etiquetas personalizadas en las descripciones de una página Web. Mediante una descripción del metalenguaje XML, el significado de las etiquetas personalizadas se define en la información transmitida al sitio del cliente. Para más información sobre XML, el lector interesado deberá consultar [PAR99] y [STL99].

29.2 EL PROCESO DE IWEB

Las características de sistemas y aplicaciones basados en Web influyen enormemente en el proceso de IWeb. La inmediatez y la evolución continúan dictando un modelo de proceso incremental e interactivo (Capítulo 2) que elabora versiones de WebApps muy rápidamente. La naturaleza intensiva de red de las aplicaciones en este dominio sugiere una población de usuarios diversa (exigiendo especialmente la obtención y modelado de requisitos), y una arquitectura de aplicación que pueda ser altamente especializada (realizando de esta manera exigencias en el diseño). Dado que las WebApps suelen ser controladas por el contenido haciendo hincapié en la estética, es probable que las actividades de desarrollo paralelas se planifiquen dentro del proceso IWeb y necesiten un equipo de personas tanto técnicas como no (por ejemplo, redactores publicitarios, diseñadores gráficos).



La IWeb demanda un proceso de software incremental y evolutivo

29.3 UN MARCO DE TRABAJO PARA LA IWEB

A medida que la evolución de las WebApps pasa de utilizar recursos estáticos de información controlada por el contenido a utilizar entornos de aplicaciones dinámicos controlados por el usuario, cada vez es más importante la necesidad de aplicar una gestión sólida y unos principios de ingeniería. Para conseguir esto, es necesario desarrollar un marco de trabajo IWeb que acompañe a un modelo de proceso eficaz, popularizado por las actividades[4] del marco de trabajo y por las tareas de ingeniería. En la Figura 29.2 se sugiere un modelo de proceso para IWeb.

El proceso IWeb comienza con la *formulación* —actividad que identifica las metas y los objetivos de la WebApp y establece el ámbito del primer incremento—.

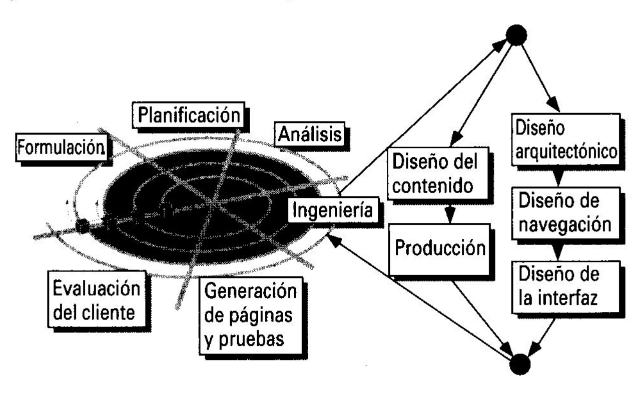


FIGURA 29.2. El modelo de proceso IWeb.

La *planificación* estima el coste global del proyecto, evalúa los riesgos asociados con el esfuerzo del desarrollo, y define una planificación del desarrollo bien granulada para el incremento final de la WebApp, con una planificación más toscamente granulada para los incrementos subsiguientes. El *análisis* establece los requisitos técnicos para la WebApp e identifica los elementos del contenido que se van a incorporar. También se definen los requisitos del diseño gráfico (estética).

La actividad de *ingeniería* incorpora dos tareas paralelas, como se muestra a la derecha en la Figura 29.2. El *diseño del contenido* y la *producción* son tareas llevadas a cabo por personas no técnicas del equipo IWeb. El objetivo de estas tareas es diseñar, producir, y/o adquirir todo el contenido de texto, gráfico y vídeo que se vayan a integrar en la WebApp. Al mismo tiempo, se lleva a cabo un conjunto de tareas de diseño (Sección 29.5).

La *generación de páginas* es una actividad de construcción que hace mucho uso de las herramientas automatizadas para la creación de la WebApp. El contenido definido en la actividad de ingeniería se fusiona con los diseños arquitectónicos, de navegación y de la interfaz para elaborar páginas Web ejecutables en HTML, XML y otros lenguajes orientados a procesos (por ejemplo, Java). Durante esta actividad también se lleva a cabo la integración con el software intermedio (*middleware*) de componentes (es decir, CORBA, DCOM o JavaBeans). Las *pruebas* ejercitan la navegación, intentan descubrir los errores de las *applets*, guiones y formularios, y ayuda a asegurar que la WebApp funcionará correctamente en diferentes entornos (por ejemplo, con diferentes navegadores).



W3C es un consorcio de industria que proporciona acceso a información WWW de interés para los ingenieros de Web, y se puede encontrar en la dirección **www.w3.org**

Cada incremento producido como parte del proceso IWeb se revisa durante la actividad de *evaluación del*

[4] Retomando el estudio de los modelos de proceso del Capítulo 2, las actividades del marco de trabajo se realizan para todas las WebApps, mientras que las tareas se adaptan al tamaño y a la complejidad de la WebApp que se va a desarrollar.

cliente. Es en este punto en donde se solicitan cambios (tienen lugar ampliaciones del ámbito). Estos cambios se integran en la siguiente ruta mediante el flujo incremental del proceso.

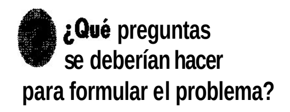
29.4 FORMULACIÓN Y ANÁLISIS DE SISTEMAS BASADOS EN WEB

La formulación y el análisis de sistemas y aplicaciones basados en Web representan una sucesión de actividades de ingeniería Web que comienza con la identificación de metas globales para la WebApp, y termina con el desarrollo de un modelo de análisis o especificación de los requisitos para el sistema. La formulación permite que el cliente o diseñador establezca un conjunto común de metas y objetivos para la construcción de la WebApp. También identifica el ámbito de esfuerzo en el desarrollo y proporciona un medio para determinar un resultado satisfactorio. El análisis es una actividad técnica que identifica los datos y requisitos funcionales y de comportamiento para la WebApp.

29.4.1. Formulación

Powell [POW98] sugiere una serie de preguntas que deberán formularse y responderse al comienzo de la etapa de formulación:

- ¿Cuál es la motivación principal para la WebApp?
- ¿Por qué es necesaria la WebApp?
- ¿Quién va a utilizar la WebApp?



La respuesta a estas preguntas deberá ser de lo **más** sucinto posible. Por ejemplo, supongamos que el fabricante de sistemas de seguridad en el hogar ha decidido establecer un sitio Web de comercio electrónico para vender sus productos directamente a los consumidores. Una frase que describiera la motivación de la WebApp podría ser la siguiente:

> HogarSeguroInc.com[5] permitirá a los consumidores configurar **y** comprar todos los componentes necesarios para instalar un sistema de seguridad en casa o en **su** comercio.

Es importante destacar que en esta frase no se ha proporcionado ningún detalle. El objetivo es delimitar la intención global del sitio Web.

Después de discutir con otros propietarios de HogarSeguro Inc., la segunda pregunta se podría contestar de la siguiente manera:

> HogarSeguroInc.com nos permitirá vender directamente a **los** consumidores, eliminando por tanto **los** costes de intermediarios, y mejorando de esta manera los márgenes de beneficios. También **nos** permitirá aumentar las ventas en un 25 **por** 100 por encima de las ventas anuales **y** nos permitirá penetrar en zonas geográficas en donde actualmente no tenemos almacenes de ventas.

Finalmente, la compañía define la demografía para la WebApp: «Los usuarios potenciales de HogarSeguroInc.com son propietarios de casas y de negocios pequeños.»

Las respuestas que se han establecido anteriormente implican metas específicas para el sitio Web HogarSeguroInc.com. En general, se identifican dos categorías [GNA99]:

- *Metas informativas:* indican la intención de proporcionar el contenido y/o información específicos para el usuario final.
- *Metas aplicables:* indican la habilidad de realizar algunas tareas dentro de la WebApp.



Poro toda WebApp deberán definirse **metas** informativos y aplicables.

En el contenido de la Web HogarSeguroInc.com, una meta informativa podría ser la siguiente:

> **El** sitio proporcionará a **los** usuarios especificaciones de un producto detallado, como descripción técnica, instrucciones de instalación e información de precios.

El examen de las respuestas anteriores llevará a poderse aplicar la afirmación de meta siguiente:

> HogarSeguroInc.com consultará al cliente sobre la instalación (es decir, sobre la casa, oficina/almacén minorista) que se va a proteger, **y** dará recomendaciones personalizadas sobre el producto **y** la configuración que se **va** utilizar.

Una vez que han identificado todas las metas aplicables e informativas se desarrolla el perfil del cliente. El perfil del usuario recoge «las características relevantes de los usuarios potenciales incluyendo antecedentes, conocimiento, preferencias e incluso más» [GNA99]. En el caso de HogarSeguroInc.com, el perfil de usuario identificará las características de un comprador típico de sistemas de seguridad (esta información sería proporcionada por el departamento de marketing de HogarSeguroInc.com).

Una vez que se han desarrollado las metas y los perfiles de usuarios, la actividad de formulación se centra en

[5] *HogarSeguro* ya se ha utilizado anteriormente como ejemplo en el libro.

la afirmación del ámbito para la WebApp (Capítulo 5). En muchos casos, las metas ya desarrolladas se integran en la afirmación del ámbito. Además es Útil, no obstante, indicar el grado de integración que se espera para la WebApp. **Es** decir, a menudo es necesario integrar los sistemas de información existentes (por ejemplo, la aplicación de base de datos existente) en un planteamiento basado en Web. En este punto se tienen en consideración también los temas de conectividad.

29.4.2. Análisis

Los conceptos y principios que se trataron para el análisis de los requisitos del software (Capítulo 11) se aplican sin revisión en la actividad de análisis de ingeniería Web. Para crear un modelo de análisis completo para la WebApp se elabora el ámbito definido durante la actividad de formulación. Durante la IWeb se realizan cuatro tipos de análisis diferentes:

Análisis del contenido. Se trata de la identificación del espectro completo de contenido que se va a proporcionar. En el contenido se incluyen datos de texto, gráficos, imágenes, vídeo y sonido. Para identificar y describir cada uno de los objetos de datos que se van a utilizar dentro de la WebApp se puede utilizar el modelado de datos (Capítulo 12).

Análisis de la interacción. Se trata de la descripción detallada de la interacción del usuario y la WebApp. Para proporcionar descripciones detalladas de esta interacción se pueden desarrollar casos prácticos (Capítulo 11).

Análisis funcional. Los escenarios de utilización (casos de uso) creados como parte del análisis de interacción definen las operaciones que se aplicarán en el contenido de la WebApp e implicarán otras funciones de procesamiento. Aquí se realiza una descripción detallada de todas las funciones y operaciones.

Cita:

Los buenos productos de conocimiento [WebApps] permiten que los clientes cumplan mejor sus necesidades, más rápidamente y de forma más barata por sí mismos, y no a través de usuarios finales trabajando como empleados. La habilidad de Internet de conectar clientes directamente con compañías proporciona una infraestructura para productos del conocimiento.

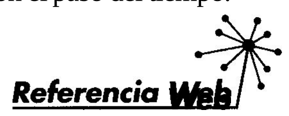
Mark McDonald

Análisis de la configuración. Se efectúa una descripción detallada del entorno y de la infraestructura en donde reside la WebApp. La WebApp puede residir en Internet, en una intranet o en una Extranet. Además, se deberá identificar la infraestructura (es decir, la infraestructura de los componentes y el grado de utilización de la base de datos para generar el contenido) de la WebApp.

Aun cuando se recomienda una especificación detallada de los requisitos para WebApps grandes y complejas, tales documentos no son los usuales. Se puede decir que la continua evolución de los requisitos de la WebApp puede hacer que cualquier documento se quede obsoleto antes de finalizarse. Aunque se puede decir que esto sucede de verdad, es necesario definir un modelo de análisis que pueda funcionar como fundamento de la siguiente actividad de diseño. Como mínimo, la información recogida durante las cuatro tareas de análisis anteriores deberá ser revisada, modificada a petición, y organizada para formar un documento que pueda pasarse a los diseñadores de WebApps

29.5 DISEÑO PARA APLICACIONES BASADAS EN WEB

La naturaleza de inmediatez de las aplicaciones basadas en Web unida a la presión de evolucionar continuamente obliga a que un ingeniero establezca un diseño que resuelva el problema comercial inmediato, mientras que al mismo tiempo obliga a definir una arquitectura de aplicación que tenga la habilidad de evolucionar rápidamente con el tiempo. El problema, desde luego, es que resolver (rápidamente) el problema inmediato puede dar como resultado compromisos que afectan a la habilidad que tiene la aplicación de evolucionar con el paso del tiempo.



Referencia Web

Una fuente valiosa de líneas generales prácticas para el diseño de sitios Web se puede encontrar en **www.ibm.com/ibm/easy/design/lower/f060100.html**

Con objeto de realizar un diseño eficaz basado en Web, el ingeniero deberá trabajar reutilizando cuatro elementos técnicos [NAN98]:

Principios y métodos de diseño. Es importante destacar que los conceptos y principios del diseño estudiados en el Capítulo **13** se aplican a todas las WebApps. La modularidad eficaz (exhibida con una cohesión alta y con un acoplamiento bajo), la elaboración paso a paso, y cualquier otra heurística de diseño del software conducirá a sistemas y aplicaciones basados en Web más fáciles de adaptar, mejorar, probar y utilizar.

Cuando se crean aplicaciones Web se pueden reutilizar **los** métodos de diseño que se utilizan para los sistemas orientados a objetos estudiados anteriormente en este libro. La hipermedia define «objetos» que interactúan mediante un protocolo de comunicación algo similar a la mensajería. De hecho, la notación de diagramas

propuesta por UML (Capítulos 21 y 22) puede adaptarse y utilizarse durante las actividades de diseño de las WebApps. Además, se han propuesto otros hipermedios de métodos de diseño (por ejemplo, [ISA95], [SCH96]).

Cita:

Para algunos, el diseño de Web se centra en la apariencia visual e interacción.... Para otros, se trata de estructurar la información y la navegación por el espacio del documento. Otros podrían aún considerar el diseño Web como la tecnología utilizada para construir aplicaciones Web. En realidad, el diseño incluye todas estas cosas y quizás más.

Thomas Powell

Reglas de oro. Las aplicaciones hipermedia interactivas (WebApps) llevan construyéndose ya hace una década. Durante ese tiempo, los diseñadores han desarrollado un conjunto de heurísticas de diseño *(reglas de oro)* que se podrán volver a aplicar durante el diseño de aplicaciones nuevas.

Configuraciones de diseño. Como se ha destacado anteriormente en este libro, las configuraciones de diseño son un enfoque genérico para resolver pequeños problemas que se pueden adaptar a una variedad más amplia de problemas específicos. En el contexto de las WebApps, las configuraciones de diseño se pueden aplicar no solo a los elementos funcionales de una aplicación, sino también a los documentos, gráficos y estética general de un sitio Web.

Plantillas. Las plantillas se pueden utilizar para proporcionar un marco de trabajo esquemático de cualquier configuración de diseño o documento a utilizar dentro de una WebApp. Nanard y Kahn [NAN98] describen este elemento de diseño reutilizable de la siguiente manera:

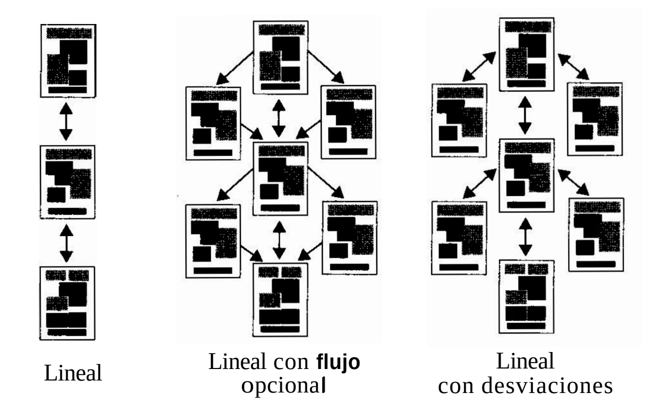


FIGURA **29.3.** Estructuras lineales.

> Una vez que se ha especificado una plantilla, cualquier parte de una estructura hipermedia que se acopla a esta plantilla se podrá generar o actualizar automáticamente llamando solamente a la plantilla con datos relevantes [para dar cuerpo al esquema]. La utilización de plantillas constructivas depende implícitamente del contenido separado de los documentos hipermedia, de la especificación y de su presentación: los datos fuente se organizan en la estructura del hipertexto tal y como se especifica en la plantilla

29.5.1. Diseño arquitectónico

El diseño arquitectónico para los sistemas y aplicaciones basados en Web se centra en la definición de la estructura global hipermedia para la WebApp, y en la aplicación de las configuraciones de diseño y plantillas constructivas para popularizar la estructura (y lograr la reutilización). Una actividad paralela, llamada *diseño del contenido*[6], deriva la estructura y el formato detallados del contenido de la información que se presentará como parte de la WebApp.



la mayoría de las estructuras de las WebApps simplemente aparecen. Evite esto trampa. Establezca el diseña estructural de la WebApp explícitamente antes de empezar a desarrollar los detalles de la página y de la navegación.

Estructuras de las WebApps

La estructura arquitectónica global va unida a las metas establecidas para una WebApp, al contenido que se va a presentar, a los usuarios que la visitarán y a la filosofía de navegación (Sección 29.5.3) establecidos. Cuando el encargado de la arquitectura va a realizar el diseño de una WebApp típica puede elegir entre cuatro fuentes diferentes [POW98].

¿Cuáles son las opciones disponibles para el diseño de una WebApp?

Las *estructuras lineales* (Fig. 29.3) aparecen cuando es común la sucesión predecible de interacciones (con alguna variación o diversificación). Un ejemplo clásico podría ser la presentación de un manual de usuario en la que las páginas de información se presentan con gráficos relacionados, vídeos cortos o sonido solo después de haber presentado un prerrequisito. La sucesión de presentación del contenido queda predefinida y se puede decir que, generalmente, es lineal. Otro ejemplo podría ser la sucesión de una entrada de pedido de un producto donde se tenga que especificar la información específica en un orden específico. En tales casos,

[6] El diseño del contenido es una actividad no técnica que llevan a cabo redactores publicitarios, artistas, diseñadores gráficos y otros que generan el contenido de las WebApps. Para más información, véase [DIN98] y [LYN99].

las estructuras que se muestran en la Figura 29.3 son las adecuadas. A medida que el contenido y el procesamiento crecen en complicación, el flujo puramente lineal que se muestra a la derecha da como resultado estructuras lineales más sofisticadas en las que se puede invocar el contenido alternativo, o en donde tiene lugar una desviación para adquirir un contenido complementario (la estructura se muestra a la derecha en la Fig. 29.3).

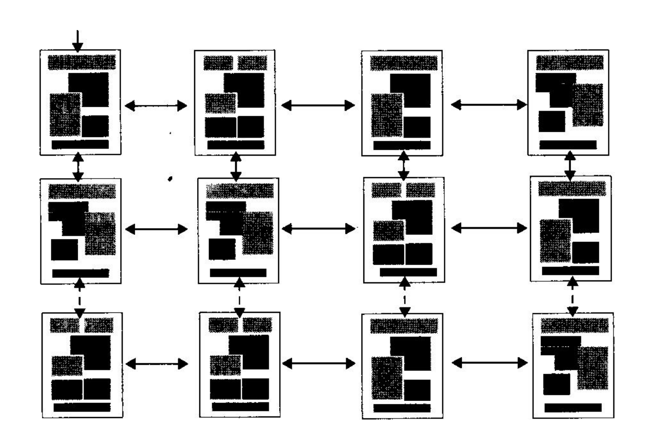


FIGURA 29.4. Estructura reticular.

Las *estructuras reticulares* (Fig. 29.4) son una opción arquitectónica que puede aplicarse cuando el contenido de la WebApp puede ser organizado categóricamente en dos dimensiones (o más). Por ejemplo, consideremos una situación en la que un sitio de comercio electrónico vende palos de golf. La dimensión horizontal de la retícula representa el tipo de palo en venta (por ejemplo, maderas, hierros, wedges, putters). La dimensión vertical representa la oferta proporcionada por los fabricantes de palos de golf. De aquí que un usuario pueda navegar por la retícula horizontalmente para encontrar la columna de los putters, y recorrer la columna para examinar las ofertas proporcionadas por los vendedores de putters. Esta arquitectura WebApp es de utilidad solo cuando se encuentra un contenido muy regular [POW98].



Las estructuras reticulares funcionan bien cuando el contenido puede organizarse categóricamente en dos dimensiones o más.

Las *estructuras jerárquicas* (Fig. 29.5) son sin duda la arquitectura WebApp más común. A diferencia de la división de jerarquías de software estudiadas en el Capítulo 14, que fomentan el flujo de control solo a lo largo de las ramas verticales de la jerarquía, se podrá diseñar una estructura jerárquica de la WebApp para posibilitar (por medio de la ramificación de hipertexto) el flujo de control en horizontal atravesando las ramas verticales de la estructura. Por tanto, el contenido presentado en la rama del extremo izquierdo de la jerarquía puede tener enlaces de hipertexto que lleven al contenido que existe en medio de la rama derecha de la estructura. Sin embargo, debería de destacarse que aunque dicha rama permita una navegación rápida por el contenido de la WebApp, puede originar también confusión por parte del usuario.



El acoplamiento es un temo engañoso poro lo arquitectura de los WebApps. Por un lodo, facilita lo navegación. Pero, por otro, también puede hacer que el usuario «se pierdo». No rehogo conexiones horizontales dentro de uno jerarquía.

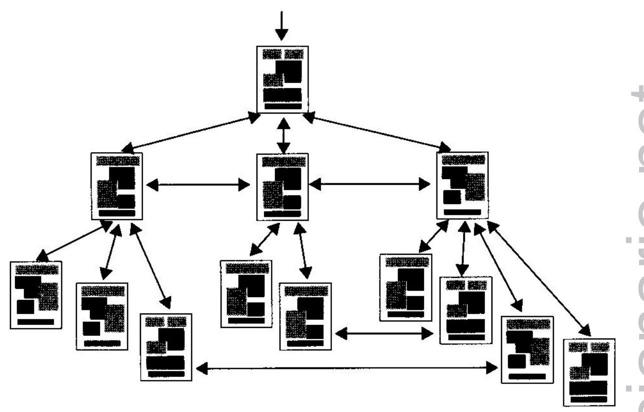


FIGURA 29.5. Estructuras jerárquicas.

Una *estructura en red o* de «web pura» (Fig. 29.6) se asemeja en muchos aspectos a la arquitectura en evolución para los sistemas orientados a objetos. Los componentes arquitectónicos (en este caso las páginas Web) se diseñan de forma que pueden pasar el control (mediante enlaces de hipertexto) a otros componentes del sistema. Este enfoque permite una flexibilidad de navegación considerable, aun cuando puede resultar confuso para el usuario.

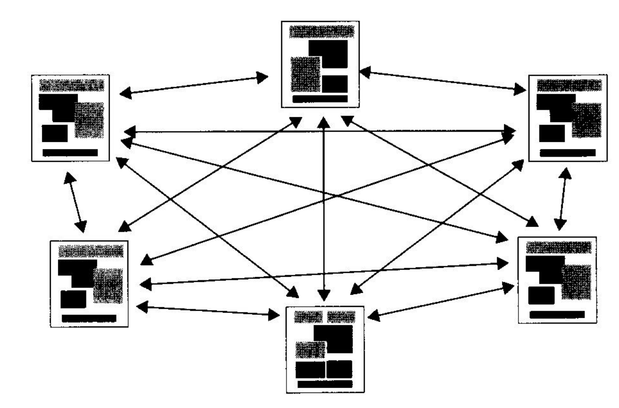


FIGURA 29.6. Estructura en red o «web pura)).

Las estructuras arquitectónicas estudiadas en los párrafos anteriores se pueden combinar para formar estruc-

turas compuestas. La arquitectura global de una WebApp puede ser jerárquica, pero parte de la estructura puede exhibir características lineales, mientras que otra parte de la arquitectura puede confeccionarse en red. La meta del diseñador arquitectónico es hacer corresponder la estructura de la WebApp con el contenido que se va a presentar y con el procesamiento que se va a llevar a cabo.

Patrones de diseño

Como se ha destacado anteriormente en este mismo libro, los patrones de diseño son un buen método para resolver pequeños problemas que pueden adaptarse a una variedad mucho más amplia de problemas específicos. En el contexto de los sistemas y aplicaciones basados en Web, los patrones de diseño pueden aplicarse en el nivel jerárquico, nivel de componentes y nivel de hipertexto (de navegación).

Referencia cruzada

En los Capítulos 14 y 22 se puede encontrar más información sobre las configuraciones de diseño.

Cuando dentro de una WebApp se requiere la funcionalidad del proceso de datos, pueden aplicarse los patrones de diseño arquitectónicas a nivel de componentes propuestas por [BUS96], [GAM95] y otros. Los patrones de diseño a nivel de hipertexto se centran en el diseño de las características de navegación que permiten al usuario moverse por el contenido de la WebApp fácilmente. Entre muchos de los patrones de diseño de hipertexto propuestos en literatura sobre este tema se encuentran los siguientes [BER98]:

- *Ciclo:* una configuración que devuelve al usuario al nodo de contenido visitado anteriormente.

Cita:

Cada una de las configuraciones es una regla compuesta de tres partes, y que expresa una relación entre un cierto contexto, un problema y una solución.
Christopher Alexander

- *Anillo de Web:* una configuración que implementa un «gran ciclo que enlaza hipertextos enteros viajando por un tema» [BER98].
- *Contorno:* un patrón que aparece cuando varios ciclos inciden en otro, permitiendo navegar por rutas definidas por los ciclos.
- *Contrapunto:* un patrón que añade comentarios de hipertexto interrumpiendo la narrativa del contenido para proporcionar más información o más indagación.
- *Mundo de espejo:* el contenido se presenta utilizando diferentes hilos narrativos, cada uno con un punto de vista o perspectiva diferente. Por ejemplo, el contenido que describe una computadora personal podría permitir al usuario seleccionar una narrativa «técnica» o «no técnica» que describa la máquina.
- *Tamiz:* una configuración que va guiando al usuario a través de una serie de opciones (decisiones) con el fin de llevar al usuario a un contenido específico e indicado por la sucesión de opciones elegidas o decisiones tomadas.
- *Vecindario:* una configuración que abarca un marco de navegación uniforme por todas la páginas Web para permitir que un usuario tenga una guía de navegación consecuente independientemente de la localización de la WebApp.

Las configuraciones de diseño de hipertexto que se han descrito anteriormente se pueden reutilizar a medida que el contenido va adquiriendo el formato que permitirá la navegación a través de una WebApp.

29.5.2. Diseño de navegación

Una vez establecida una arquitectura de WebApp, una vez identificados los componentes (páginas, guiones, applets y otras funciones de proceso) de la arquitectura, el diseñador deberá definir las rutas de navegación que permitan al usuario acceder al contenido y a los servicios de la WebApp. Para que el diseñador pueda llevarlo a cabo, debe (1) identificar la semántica de la navegación para diferentes usuarios del sitio; y (2) definir la mecánica (sintaxis) para lograr la navegación.

Generalmente una WebApp grande tendrá una variedad de roles de usuarios diferentes. Por ejemplo, los roles podrían ser el de *visitante, cliente registrado o cliente privilegiado.* Cada uno de estos roles se pueden asociar a diferentes niveles de acceso al contenido y de servicios diferentes. Un *visitante* puede tener acceso sólo a un contenido limitado, mientras que un *cliente registrado* puede tener acceso a una variedad mucho más amplia de información y de servicios. La semántica de la navegación de cada uno de estos roles sería diferente.

El diseñador de WebApps crea una *unidad semántica de navegación* (USN) para cada una de las metas asociadas a cada uno de los roles de usuario [GNA99]. Por ejemplo, un *cliente registrado* puede tener seis metas diferentes, todas ellas con un acceso a información y servicios diferentes. Para cada meta se crea una USN. Gnaho y Larcher [GNA99] describen la USN de la siguiente manera:

> La estructura de una USN se compone de un conjunto de subestructuras de navegación que llamamos *formas de navegación* [WoN *(Ways of navigating)]*. Una WoN representa la mejor forma de navegación o ruta para que usuarios con ciertos perfiles logren su meta o submeta deseada. Por tanto, el concepto de WoN se asocia al concepto de Perfil de Usuario.

> La estructura de una WoN se compone de un conjunto de *nodos de navegación* (NN) relevantes conectados a *enlaces de navegación,* entre los que algunas veces se incluyen las USNs. Eso significa que las USNs pueden agregarse para formar una USN de nivel superior, o anidarse en cualquier nivel de profundidad.



Una USN se compone de un conjunto de subestructuras de navegación llamadas ((formas de navegación)) (WoN). Una USN representa una meta de navegación específica para un tipo específico de usuario.

Durante las etapas iniciales del diseño de navegación, para determinar una o más WoNs para cada meta de usuario, se evaluará la estructura de la WebApp (arquitectura y componentes). Como se ha destacado anteriormente, una WoN identifica los nodos de navegación (por ejemplo, páginas Web), y entonces los enlaces que hacen posible la navegación entre ellos. La WoN entonces se organiza en USNs.

A medida que avanza el diseño, se va identificando la mecánica de cada enlace de navegación. Entre otras muchas opciones se encuentran los enlaces basados en texto, iconos, botones, interruptores y metáforas gráficas. El diseñador deberá elegir los enlaces de navegación adecuados para el contenido y consecuentes con la heurística que conduce al diseño de una interfaz de alta calidad.

Además de elegir la mecánica de navegación, el diseñador también deberá establecer las convenciones y ayudas adecuadas. Por ejemplo, los iconos y los enlaces gráficos deberán tener un aspecto «clickable (capacidad de accederse)» con los bordes biselados, y dar así una imagen en tres dimensiones. La realimentación visual o de sonido se deberá diseñar para proporcionar al usuario una indicación de que se ha elegido una opción de navegación. Para la navegación basada en texto, se deberá utilizar el color que indica los enlaces de navegación y proporcionar una indicación de los enlaces por los que se ha navegado. Estas son solo unas pocas convenciones de las docenas que hacen que la navegación por la red sea agradable. Además de las convenciones, en este punto también se deberán diseñar ayudas de navegación tales como mapas de sitios, tablas de contenidos, mecanismos de búsqueda y servicios dinámicos de ayuda

29.5.3. Diseño de la interfaz

Los conceptos, principios y métodos de diseños de interfaz que se presentaron en el Capítulo 15 son todos aplicables al diseño de interfaces de usuario para WebApps. Sin embargo, las características especiales de los sistemas y aplicaciones Web requieren otras consideraciones adicionales.

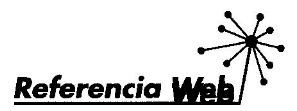
Cita:
Los usuarios tienen poca paciencia con los sitios WWW que tienen un diseño pobre.
Jakob Nielsen y Annette Wagner

La interfaz de usuario de una WebApp es la «primera impresión». Independientemente del valor del contenido, la sofisticación de las capacidades, los servicios de procesamiento y el beneficio global de la WebApp en sí, una interfaz con un diseño pobre decepcionará al usuario potencial y podrá de hecho hacer que el usuario se vaya a cualquier otro sitio. Dado el gran volumen de WebApps que compiten virtualmente en todas las áreas temáticas, la interfaz debe «arrastrar» inmediatamente al usuario potencial. Nielsen y Wagner [NIE96] sugieren unas cuantas líneas generales y sencillas en el rediseño de una WebApp:

- Probabilidad de que los errores del servidor, incluso los más pequeños, hagan que el usuario abandone el sitio Web y busque información y servicios en algún otro sitio.

¿Cuáles son algunas de las líneas generales básicas para el diseño de la interfaz de un sitio Web?

- La velocidad de lectura del monitor de una computadora es aproximadamente un 25 por 100 más lento que leer una copia impresa. Por tanto, no hay que obligar al usuario a leer cantidades voluminosas de texto, particularmente cuando el texto explica la operación de la WebApp o ayuda a navegar por la red.
- Evite los símbolos «bajo construcción» —levantan expectación y provocan un enlace innecesario que seguramente sea decepcionante—.
- Los usuarios prefieren no tener que recorrer la pantalla. Dentro de las dimensiones normales de una ventana del navegador se deberá incluir información importante.
- Los menús de navegación y las barras de cabecera se deberán diseñar consecuentemente y deberán estar disponibles en todas las páginas a las que el usuario tenga acceso. El diseño no deberá depender de las funciones del navegador para ayudar en la navegación.
- La estética nunca deberá sustituir la funcionalidad. Por ejemplo, un botón sencillo podría ser una opción de navegación mejor que una imagen o icono estéticamente agradables, pero vagos cuya intención no es muy clara.



Uno de los mejores grupos de recursos de usabilidad de la Web ha sido recogido por el Grupo de Usabilidad, y se puede encontrar en la dirección **www.usability.com/umi_links.htm**

- Las opciones de navegación deberán ser obvias, incluso para el usuario casual. El usuario deberá buscar la pantalla para determinar cómo enlazar con otro contenido o servicio.

Una interfaz bien diseñada mejora la percepción del contenido o de los servicios del usuario que proporciona el sitio Web. No tiene que ser necesariamente deslumbrante, pero deberá estar siempre bien estructurada y ergonómica. Un estudio completo de las interfaces WebApp de usuario está fuera del ámbito de este libro. Los lectores interesados pueden consultar [SAN96], [FLE98], [ROS98], o [LYN99] entre los cientos de ofertas que existen como guía.

29.6 PRUEBAS DE LAS APLICACIONES BASADAS EN WEB

En el Capítulo 17, se destacó que las pruebas son el proceso de ejercitar el software con la intención de encontrar (y por último corregir) los errores. Esta filosofía fundamental no se cambiará para el caso de las WebApps. De hecho, dado que los sistemas y aplicaciones basados en Web residen en una red e interoperan con muchos sistemas operativos diferentes, navegadores, plataformas de hardware, y protocolos de comunicación, la búsqueda de errores representa un reto significativo para los ingenieros Web.

¿Qué pasos hay que seguir para la comprobación de una WebApp?

El enfoque de las pruebas de las WebApps adopta los principios básicos de todas las pruebas del software (Capítulo 17) y aplica estrategias y tácticas que ya han sido recomendadas para los sistemas orientados a objetos (Capítulo 23). Este enfoque se resume en los pasos siguientes:

1. *El modelo de contenido de la WebApp es revisado para descubrir errores.* Esta actividad de «prueba» se asemeja en muchos aspectos a la de un corrector ortográfico de un documento escrito. De hecho, un sitio Web grande tendrá la capacidad de construir un listado de los servicios de correctores profesionales para descubrir errores tipográficos, errores gramaticales, errores en la consistencia del contenido, errores en representaciones gráficas y de referencias cruzadas.



La innovación es un negocio amargo para los que **comprueban el software. Cuando ya** se sabe qué **pruebas aplicar en una tecnología** en particular, **aparece una nueva (Internet y WebApps)**, y se **vienen abajo los anteriores.**
James Bach

2. *El modelo de diseño para la WebApp es revisado para descubrir errores de navegación.* Los casos prácticos derivados como parte de la actividad de análisis permite que un ingeniero Web ejercite cada escenario de utilización frente al diseño arquitectónico y de navegación. En esencia, estas pruebas no ejecutables ayudan a descubrir errores en la navegación (por ejemplo, un caso en donde el usuario no pueda leer un nodo de navegación). Además, los enlaces de navegación (Sección 29.5.2) son revisados para asegurar su correspondencia con los especificados en cada USN del rol de usuario.
3. *Se aplican pruebas de unidad a los componentes de proceso seleccionados y las páginas Web.* Cuando lo que se tiene en consideración es el tema de las WebApps el concepto de unidad cambia. Cada una de las páginas Web encapsulará el contenido, los enlaces de navegación y los elementos de procesamiento (formularios, guiones, applets). No siempre es posible o práctico comprobar cada una de las características individualmente. En muchos casos, la unidad comprobable más pequeña es la página Web. A diferencia de la comprobación de unidades de software convencional, que tiende a centrarse en el detalle algorítmico de un módulo y los datos que fluyen por la interfaz del módulo, la comprobación por páginas se controla mediante el contenido, proceso y enlaces encapsulados por la página Web.

Referencia cruzada
Las estrategias de **integración se** abarcan en los **Capítulos** 18 y **23.**

4. *Se construye la arquitectura, se realizan las pruebas de integración.* La estrategia para la prueba de integración depende de la arquitectura que se haya elegido para la WebApp. Si la WebApp se ha diseñado con una estructura jerárquica lineal, reticular o sencilla, es posible integrar páginas Web de una manera muy similar a como se integran los módulos del software convencional. Sin embargo, si se utiliza una jerarquía mezclada o una arquitectura de red (Web), la prueba de integración es similar al enfoque utilizado para los sistemas OO. La comprobación basada en hilos (Capítulo 23) se puede utilizar para integrar un conjunto de páginas Web (se puede utilizar una USN para definir el conjunto adecuado) que se requiere para responder a un suceso de usuario. Cada hilo se integra y se prueba individualmente. La prueba de regresión se aplica para asegurar que no haya efectos secundarios. La comprobación de agrupamientos integra un conjunto de páginas colaborativas (determinadas examinando los casos prácticos y la USN). Los casos de prueba se derivan para descubrir errores en las colaboraciones.
5. *La WebApp ensamblada se prueba para conseguir una funcionalidad global y un contenido.* Al igual que la validación convencional, la validación de los

sistemas y aplicaciones basados en Web se centra en acciones visibles del usuario y en salidas reconocibles para el usuario que procedan del sistema. Para ayudar en la derivación de las pruebas de validación, las pruebas deberán basarse en casos prácticos. El caso práctico proporciona un escenario con una probabilidad alta de descubrir errores en los requisitos de interacción del usuario.

6. *La WebApp se implementa en una variedad de configuraciones diferentes de entornos* y *comprobar así la compatibilidad con cada configuración.* Se crea una matriz de referencias cruzadas que define todos los sistemas operativos probables, plataformas de hardware para navegadores[7] y protocolos de comunicación. Entonces se llevan a cabo pruebas para descubrir los errores asociados con todas y cada una de las configuraciones posibles.

7. *La WebAppse comprueba con una población de usuarios finales controlada* y *monitorizada.* Se selecciona un grupo de usuarios que abarque todos los roles posibles de usuarios. La WebApp se pone en práctica con estos usuarios y se evalúan los resultados de su interacción con el sistema para ver los errores de contenido y de navegación, los intereses en usabilidad, compatibilidad, fiabilidad y rendimiento de la WebApp.

Dado que muchas WebApps están en constante evolución, el proceso de comprobación es una actividad continua, dirigida por un personal de apoyo a la Web que utiliza pruebas de regresión derivadas de pruebas desarrolladas cuando se creó la WebApp.

29.7 PROBLEMAS DE GESTIÓN

Dada la inmediatez de las WebApps, sería razonable preguntarse: ¿necesito realmente invertir tiempo esforzándome con la WebApp? ¿no debería dejarse que la WebApp evolucionara de forma natural, con poca o ninguna gestión explícita? Muchos diseñadores de Webs optarían por gestionar poco o nada, pero eso no hace que estén en lo cierto.

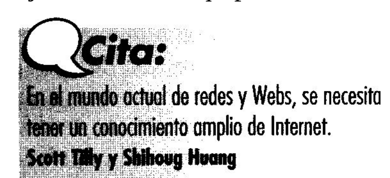
La ingeniería Web es una actividad técnica complicada. Hay muchas personas implicadas, y a menudo trabajando en paralelo. La combinación de tareas técnicas y no técnicas que deben de tener lugar (a tiempo y dentro del presupuesto) para producir una WebApp de alta calidad representa un reto para cualquier grupo de profesionales. Con el fin de evitar cualquier confusión, frustración y fallo, se deberá poner en acción una planificación, tener en cuenta los riesgos, establecer y rastrear una planificación temporal y definir los controles. Estas son las actividades clave que constituyen lo que se conoce como gestión de proyectos.

Referencia cruzada

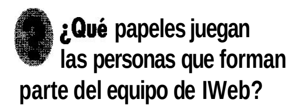
Las actividades asociadas con la gestión de proyectos de software se estudian en la Parte Segunda de este libro.

29.7.1. El equipo de IWEB

La creación de una buena aplicación Web exige un amplio abanico de conocimientos. Tilley y Huang [TIL99] se enfrentan a este tema cuando afirman: «Hay muchos aspectos diferentes en relación con el software de aplicaciones [Web] debido al resurgimiento del renacentista, aquel que se encuentra cómodo trabajando en varias disciplinas...» Aun cuando los autores están absolutamente en lo cierto, los «renacentistas» no disponen de muchas provisiones, y dado las exigencias asociadas a los proyectos importantes de desarrollo de WebApps, el conjunto de conocimientos diversos necesario podrá distribuirse incluso mejor dentro del equipo de IWeb.



Los equipos de IWeb pueden organizarse de forma muy similar a como se organizan los equipos de software del Capítulo **3.** Sin embargo, pueden existir diferencias entre los participantes y sus roles. Entre los muchos conocimientos que deben distribuirse por los miembros del equipo IWeb se encuentran los siguientes: ingeniería del software basada en componentes, realización de redes, diseño arquitectónico y de navegación, lenguajes y estándares de Intemet, diseño de interfaces para personas, diseño gráfico, disposición del contenido y pruebas de las WebApps. Los roles[8] siguientes deberán ser distribuidos entre los miembros del equipo IWeb:



Desarrolladores y proveedores de contenido. Dado que las WebApps son controladas inherentemente por el contenido, el papel de los miembros del equipo IWeb se centra en la generación y/o recolección del conteni-

[7] Los navegadores son excelentes para implementar sus propias interpretaciones «estándar» ligeramente diferentes de HTML y Javascript. Para un estudio de temas de compatibilidad, véase **www.browsercaps.com.**

[8] Estos roles se han adaptado de Harsen y sus colaboradores [HAN99].

do. Retomando la idea de que el contenido abarca un amplio abanico de objetos de datos, los diseñadores y proveedores del contenido pueden proceder de diversos planos de fondo (no de software). Por ejemplo, el personal de ventas o de marketing puede proporcionar información de productos e imágenes gráficas: los productores de medios pueden proporcionar imagen y sonido; los diseñadores gráficos pueden proporcionar diseños para composiciones gráficas y contenidos estéticos; **los** redactores publicitarios pueden proporcionar contenido basado en texto. Además, existe la posibilidad de necesitar personal de investigación que encuentre y dé formato al contenido externo y lo ubique y referencie dentro de la WebApp.

Editores de Web. El contenido tan diverso generado por los desarrolladores y proveedores de contenido deberá organizarse e incluirse dentro de la WebApp. Además, alguien deberá actuar como conexión entre el personal técnico que diseña la WebApp y los diseñadores y proveedores del contenido. Esta función la lleva a cabo el editor de Web, quien deberá entender la tecnología tanto del contenido como de la WebApp en donde se incluye el lenguaje HTML (o ampliaciones de generaciones posteriores, tal como XML), la funcionalidad de bases de datos y guiones, y la navegación global del sitio Web.

Ingeniero de Web. Un ingeniero Web cada vez se involucra más con la gran cantidad de actividades del desarrollo de una WebApp entre las que se incluyen la obtención de requisitos, modelado de análisis, diseño arquitectónico, de navegación y de interfaces, implementación de la WebApp **y** pruebas. El ingeniero de Web también deberá conocer a fondo las tecnologías de componentes, las arquitecturas cliente/servidor, HTML/XML, **y** las tecnologías de bases de datos; y deberá tener conocimiento del trabajo con conceptos de multimedia, plataformas de hardware/software, seguridad de redes y temas de soporte a los sitios Web.

Especialistas de soporte. Este papel se asigna a la persona (personas) que tiene la responsabilidad de continuar dando soporte a la WebApp. Dado que las WebApps están en constante evolución, el especialista de soporte es el responsable de las correcciones, adaptaciones y mejoras del sitio Web, donde se incluyen actualizaciones del contenido, implementación de productos y formularios nuevos, y cambios del patrón de navegación.

Administrador. Se suele llamar *Web master,* y es el responsable del funcionamiento diario de la WebApp, en donde se incluye:

- el desarrollo e implementación de normas para el funcionamiento de la WebApp;
- el establecimiento de los procedimientos de soporte y realimentación;
- los derechos de acceso y seguridad de la implementación;
- la medición y análisis del tráfico del sitio Web;
- la coordinación de los procedimientos de control de cambios (Sección 29.7.3);
- la coordinación con especialistas de soporte.

El administrador también puede estar involucrado en las actividades técnicas realizadas por los ingenieros de Web y por los especialistas de soporte.

29.7.2. Gestión del proyecto

En la Parte Segunda de este libro se tuvieron en consideración todas y cada una de las actividades globalmente llamadas gestión de proyectos[9]. Las métricas de procesos y proyectos, la planificación de proyectos (y estimación), el análisis y gestión de riesgos, la planificación temporal y el rastreo, SQA y CGS, se estudiaron en detalle. En teoría, la mayoría de las actividades de gestión de proyectos, si no todas, estudiadas en capítulos anteriores, se aplican a los proyectos de IWeb. Pero en la práctica, el enfoque de IWeb para la gestión de proyectos es considerablemente diferente.

En primer lugar, se subcontrata un porcentaje sustancial[10] de WebApps a proveedores que (supuestamente) son especialistas en el desarrollo de sistemas y aplicaciones basados en Web. En tales casos, un negocio (el cliente) solicita un precio fijo para el desarrollo de una WebApp a uno o más proveedores, evalúa los precios de los candidatos y selecciona un proveedor para hacer el trabajo. Pero, ¿qué es lo que busca la organización contratista?, ¿cómo se determina la competencia de un proveedor de WebApps?, ¿cómo se puede saber si un precio es razonable?, ¿qué grado de planificación, programación temporal, y valoración de riesgos se puede esperar a medida que una organización (y **su** subconstratista) se va embarcando en un desarrollo importante de WebApps?

En segundo lugar, el desarrollo de WebApps es una área de aplicación relativamente nueva por tanto se pueden utilizar pocos datos para la estimación. Hasta la fecha, no se ha publicado virtualmente ninguna métrica de IWeb. De hecho, tampoco han surgido muchos estudios sobre lo que esas métricas podrían ser. Por tanto, la estimación es puramente cualitativa —basada en

[9] Se recomienda que lectores no familiarizados con los conceptos básicos de gestión de proyectos revisen en este momento el Capítulo 3.

[10] Aun cuando los datos de industria fiables son dificiles de encontrar, es seguro decir que este porcentaje es considerablemente mayor que el que se encuentra en el trabajo del software convencional.

experiencias anteriores con proyectos similares—. Pero casi todas las WebApps tienen que ser innovadoras —ofreciendo algo nuevo y diferente a aquellos que la utilizan—. De aquí que la estimación experimental, aunque sea Útil, está abierta a errores considerables. Por consiguiente, ¿cómo se derivan estimaciones fiables?, ¿con qué grado de seguridad pueden cumplirse los programas temporales definidos?

En tercer lugar, el análisis de riesgos y la programación temporal se predican bajo un entendimiento claro del ámbito del proyecto. Y sin embargo, la característica de «evolución continua» tratada en la Sección 29.1 sugiere que el ámbito de la WebApp sea fluido. ¿Cómo pueden controlar los costes la organización contratista y el proveedor subcontratado?, y ¿cómo pueden planificar el momento en que es probable que los requisitos cambien drásticamente a lo largo del proyecto?, ¿cómo se puede controlar el cambio del ámbito?, y lo que es más importante, dado su naturaleza ¿deberán controlarse los sistemas y aplicaciones basados en Web ?

Llegado a este punto de gestión de proyectos para WebApps, las preguntas que se han formulado por las diferencias destacadas anteriormente no son fáciles de responder. Sin embargo, merece la pena tener en consideración una serie de líneas generales.

Inicio de un proyecto. Aun cuando la subcontratación sea la estrategia elegida para el desarrollo de WebApps, una organización deberá realizar una serie de tareas antes de buscar el proveedor de subcontratación para hacer el trabajo.

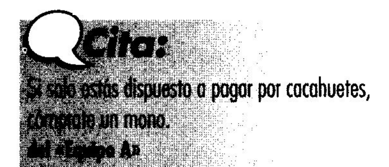
1. *Muchas de las actividades de análisis tratadas en la Sección 29.3 se deberán realizar internamente.* Se identifica el público de la WebApp; se confecciona un listado de aquellos con intereses internos en la WebApp; y se definen y revisan las metas globales de la WebApp; se especifica tanto la información como los servicios que se van a proporcionar en la WebApp; se destacan los sitios Web rivales, y se definen las «medidas» cuantitativas y cualitativas para obtener una WebApp con éxito. Esta información deberá de documentarse en una especificación del producto.
2. *Se deberá desarrollar internamente un diseño aproximado de la WebApp.* Obviamente una persona experta en el desarrollo de WebApps creará un diseño completo, pero puede ahorrar tiempo y dinero identificando el aspecto e interacción general de la WebApp para el proveedor de subcontratación (esto siempre puede modificarse durante las etapas preliminares del proyecto). El diseño deberá incluir la indicación del proceso interactivo que se va a llevar a cabo (por ejemplo, formularios, entrada de pedidos).
3. *Se deberá desarrollar una planificación temporal aproximada del proyecto, incluyendo no solo las fechas finales de entrega, sino también las fechas hito (significativas).* Los hitos deberán adjuntarse a las versiones de entrega posibles a medida que avanza el proyecto.
4. *Se deberá identificar el grado de supervisión e interacción del contratista con el proveedor.* Deberá incluirse el nombre del contacto del proveedor y la identificación de la responsabilidad y autoridad del contacto, la definición de los puntos de revisión de calidad a medida que el desarrollo va avanzando, y la responsabilidad de los proveedores en relación con la comunicación entre organizaciones.

Toda la información desarrollada en los pasos anteriores deberá de organizarse en una *solicitud de opciones* que se transmite a los proveedores candidatos[11].

Cita:

¿Que cuál es mi consejo para los constructores de [WebApps]? 1. Establecer los requisitos de calidad cuantitativamente. 2. Conseguir garantías contractuales. 3. Utilizar proveedores con fama demostrada. 4. Construir y ampliar el sistema por etapas evolutivas. 5. Tener en cuenta la redundancia sistemática adecuada a muchos niveles para permitir algún grado de operación cuando aparezcan errores.

Tom Gilb



Selección entre los proveedores de subcontratación candidatos. En los últimos años han surgido miles de compañías de «diseño Web» para ayudar a las empresas a establecer una presencia y/o implicación Web en el comercio electrónico. Muchas han llegado a tener adicción por el proceso IWeb, mientras que otras muchas se asemejan a los intrusos informáticos *(hackers)*. Con objeto de seleccionar el desarrollador de Web candidato, el contratista deberá llevar a cabo una diligencia obligada: (1) entrevistar a los clientes antiguos para determinar la profesionalidad del proveedor de Web, la habilidad de cumplir los compromisos de horarios y costes, y la habilidad de comunicarse eficazmente; (2) determinar el nombre del ingeniero jefe de Web del proveedor de proyectos anterior con éxito (y después cerciorarse de que esta persona se vea obligada mediante contrato a su implicación en el proyecto), y **(3)** examinar cuidadosamente las muestras de trabajo del proveedor simi-

[11] Si el trabajo de desarrollo de la WebApp es dirigida por un grupo interno, nada cambia. El proyecto se inicia de la misma manera.

lares en aspecto e interacción (y en área de negocios) a la WebApp que se va a contratar. Antes incluso de que se vaya a establecer la oferta, una reunión cara a cara puede proporcionar una buena impresión para que el contratista y el proveedor conecten bien.

Evaluación de la validez de las ofertas de precios y de la fiabilidad de las estimaciones. Dado que existen muy pocos datos históricos y el ámbito de las WebApps es notoriamente fluido, la estimación es inherentemente arriesgada. Por esta razón algunos proveedores incorporarán márgenes sustanciales de seguridad en las ofertas de precios de un proyecto. Evidentemente esto es comprensible y adecuado. La pregunta no es si tenemos la mejor solución para nuestra inversión, sino la serie de preguntas que se muestran a continuación:

- ¿Es el coste acordado una buena oferta para proporcionar un beneficio directo o indirecto sobre la inversión y justificar así el proyecto?
- ¿Es el proveedor de la oferta una muestra clara de la profesionalidad y experiencia requeridos?

Si la respuesta es «sí», el precio ofertado es justo.

El grado de gestión del proyecto que se puede esperar o realizar. La formalidad asociada con las tareas de gestión del proyecto (llevadas a cabo tanto por el proveedor como por el contratista) es directamente proporcional al tamaño, coste y complejidad de la WebApp. Para proyectos complejos y grandes deberá desarrollarse una planificación que defina las tareas del trabajo, los puntos de comprobación, los productos de trabajo de ingeniería, los puntos de revisión del cliente y los hitos importantes. El proveedor y el contratista deberán evaluar el riesgo en conjunto, y desarrollar los planes de mitigar, monitorizar y gestionar esos riesgos considerados como importantes. Los mecanismos para la seguridad de calidad y el control de cambios se deberán definir explícitamente al escribirse. Y tendrán que establecerse métodos de comunicación entre el contratista y el proveedor.

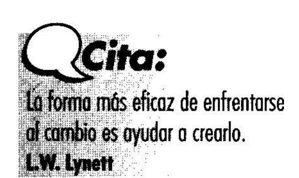
Evaluación de la planificación temporal del desarrollo. Dado que las planificaciones de desarrollo de las WebApps abarcan un período de tiempo relativamente corto (normalmente menos de un mes o dos), la planificación de desarrollo deberá tener un alto grado de granularidad. Es decir, las tareas del trabajo y los hitos menores se tendrán que planificar día a día. Esta granularidad permite que el contratista y el proveedor reconozcan la reducción del tiempo de planificación antes de que amenace la fecha de finalización.

Gestión del ámbito. Dado que es muy probable que el ámbito cambie a medida que avanza el proyecto de la WebApp, el modelo de proceso deberá ser incremental (Capítulo 2). Esto permite que el equipo de desarrollo «congele» el ámbito para poder crear una nueva versión operativa de la WebApp. El incremento siguiente puede afrontar los cambios de ámbito sugeridos por una revisión del incremento precedente, pero una vez que comienza el incremento, el ámbito se congela de nuevo «temporalmente». Este enfoque hace posible que el equipo de la WebApp trabaje **sin** tener que aceptar una corriente continua de cambios, pero reconociendo la característica de evolución continua de la mayoría de las WebApps.

Las líneas generales sugeridas anteriormente no pretenden ser un recetario a prueba de tontos para WebApps puntuales y con una producción a un coste bajo. Sin embargo, ayudarán tanto al contratista como al proveedor a iniciar el trabajo suavemente con unos conocimientos mínimos.

29.7.3. Problemas GCS para la IWEb

Durante la última década, las WebApps han evolucionado y han pasado de utilizar dispositivos informales para la difusión de información a utilizar sitios sofisticados para el comercio electrónico. **A** medida que las WebApps van creciendo en importancia para la supervivencia y el crecimiento de los negocios, también crece el control de la configuración. ¿Por qué? Porque sin controles eficaces cualquier cambio inadecuado en una WebApp (recordemos que la inmediatez y la evolución continua son los atributos dominantes de muchas WebApps) puede conducir a: (1) una ubicación no autorizada de la información del producto nuevo; (2) una funcionalidad errónea y pobremente comprobada que frustra a los visitantes del sitio Web; **(3)** agujeros de seguridad que ponen en peligro a los sistemas internos de las compañías, y otras consecuencias económicamente desagradables e incluso desastrosas.



Se pueden aplicar las estrategias generales para la gestión de la configuración del software (GCS) descritas en el Capítulo 9, pero las tácticas y herramientas deberán adaptarse y ajustarse a la naturaleza única de las WebApps. Cuando se desarrollan tácticas para la gestión de configuración de las WebApps, deberán tenerse en consideración cuatro temas [DAR99]: el contenido, las personas, la escalabilidad y la política.

Contenido. Una WebApp normal contiene una gran cantidad de contenido —texto, gráficos, applets, guiones, archivos de sonido y vídeo, elementos activos de páginas, tablas, corrientes de datos y muchos más—. El reto es organizar este mar de contenido en un conjunto razonable de objetos de configuración (Capítulo 9), y entonces establecer los mecanismos de control de configuración adecuados para estos objetos. Un enfoque será modelar el contenido de la WebApp mediante la utilización de técnicas convencionales de modelado de datos (Capítulo 11), adjuntando un conjunto de propiedades especializadas a cada objeto. La naturaleza estática y dinámica de

cada objeto y la longevidad proyectada (por ejemplo, existencia temporal y fija, o un objeto permanente) son ejemplos de las propiedades necesarias para establecer un enfoque GCS eficaz. Por ejemplo, si se cambia un objeto de contenido cada hora, se dice que tiene una longevidad temporal. Los mecanismos de control de este objeto serán diferentes (menos formales) de los que se aplican a un componente de formularios (objeto permanente).



Es esencial controlar el cambio durante los proyectos IWeb, aun cuando puede hacerse de formo exagerada. Empiece por los procedimientos de control de cambio de relativa informalidad (Capítulo 9). lo meto inicial será evitar los efectos de trinquete en cambios incontrolados.

Personas. Dado que un porcentaje significativo del desarrollo de las WebApps continúa realizándose dependiendo del caso, cualquier persona que esté implicada en la WebApp puede (y a menudo lo hace) crear el contenido. Muchos creadores de contenido no tienen antecedentes en ingeniería del software y no tienen ningún conocimiento de la necesidad de una gestión de configuración. La aplicación crece y cambia de forma incontrolada.

Escalabilidad. Las técnicas y controles aplicados a una WebApp pequeña no tienen buena escalabilidad. No es muy común que una WebApp sencilla crezca significativamente cuando se implementan las interconexiones que existen dentro de los sistemas de información, bases de datos, almacenes de datos y pasarelas de portales. A medida que crece el tamaño y la complejidad, los pequeños cambios pueden tener efectos inalcanzables y no intencionados que pueden ser problemáticos. Por tanto, el rigor de los mecanismos de control de la configuración deberán ser directamente proporcionales a la escalabilidad de la aplicación.

Política. ¿Quién es el propietario de una WebApp? Esta pregunta se argumenta en compañías grandes y pequeñas, y la respuesta tiene un impacto significativo en las actividades de gestión y control asociadas con la IWeb. En algunos casos los diseñadores de WebApps se encuentran fuera de la organización TI, dificultando posiblemente la comunicación. Para ayudar a entender la política asociada a IWeb, Dart [DAR99] sugiere las siguientes preguntas: ¿quién asume la responsabilidad de la información en el sitio Web? ¿quién asegura que los procesos de control de calidad se han llevado a cabo antes de publicar la información en el sitio Web? ¿quién es el responsable de hacer los cambios? ¿quién asume el coste del cambio? Las respuestas ayudarán a determinar qué personas dentro de la organización deberán adoptar un proceso de gestión de configuración para las WebApps.

La gestión de configuración para la IWeb está todavía en su infancia. Un proceso convencional de GCS puede resultar demasiado pesado y torpe. La gran mayoría de las herramientas GCS no tienen las características que permiten adaptarse fácilmente a la IWeb. Entre los temas que quedan por tratar se encuentran los siguientes [DAR99]:

- creación de un proceso de gestión de configuración que sea lo suficientemente hábil como para aceptar la inmediatez y la evolución continua de las WebApps;
- aplicación de mejores conceptos y herramientas de gestión de configuración para aquellos que desarrollan y no están familiarizados con la tecnología;
- suministro del soporte a los equipos de desarrollo de WebApps distribuidos;
- suministro de control en un entorno de cuasipublicación, donde el contenido cambia de forma casi continua;
- consecución de la granularidad necesaria para controlar una gran cantidad de objetos de configuración;
- incorporación de la funcionalidad de gestión de configuración en las herramientas IWeb existentes;
- gestión de cambios en objetos que contienen enlaces con otros objetos.

Estos y otros temas deberán afrontarse antes de que se disponga una gestión de configuración eficaz para la IWeb.

RESUMEN

Se puede argumentar que el impacto de los sistemas y aplicaciones basados en Web es el acontecimiento más significativo en la historia de la informática. A medida que las WebApps crecen en importancia, el enfoque de ingeniería de Web disciplinado ha empezado a evolucionar —basado en principios, conceptos, procesos y métodos que se han desarrollado para la ingeniería del software—.

Las WebApps son diferentes de otras categorías de software informático. Son intensivas de red, están gobernadas por el contenido y están en continua evolución. La inmediatez que conduce a su desarrollo, la necesidad primordial de seguridad al funcionar, y la demanda de estética y la entrega del contenido funcional son otros factores diferenciadores. Durante el desarrollo de la WebApp hay tres tecnologías que se integran con otras de ingeniería del software convencional; desarrollo basado en componentes, seguridad y lenguajes de notas estándar.

El proceso de ingeniería de Web comienza con la formulación —una actividad que identifica las metas y los

objetivos de la WebApp—. La planificación estima el coste global del proyecto, evalúa los riesgos asociados con el esfuerzo del desarrollo y define una programación temporal del desarrollo. El análisis establece requisitos técnicos e identifica los objetos del contenido que se incorporarán en la WebApp. La actividad de ingeniería incorpora dos tareas paralelas: diseño del contenido y diseño técnico. La generación de páginas es una actividad de construcción que hace un uso extenso de herramientas automatizadas para la creación de WebApps, y la comprobación ejercita la navegación de la WebApp, intentando descubrir errores en la función y el contenido, y asegurando mientras tanto que la WebApp funcione correctamente en diferentes entornos. La ingeniería de Web hace uso de un modelo de proceso iterativo e incremental porque la línea temporal del desarrollo para la WebApp es muy corta. Las actividades protectoras aplicadas durante el trabajo de la ingeniería del software —SQA, **GCS,** gestión de proyectos — se aplican a todos los proyectos de ingeniería de Web.

REFERENCIAS



[ATK97] Atkins, D., et al., ***Internet Security: Professional Reference,*** New Riders Publishing, 2.ª ed., 1997.

[BER98] Bemstein, M., «Pattems in Hypertext», ***Proc. 9th ACM Conf. Hypertext,*** ACM Press, 1998, pp. 21-29.

[BRA97] Bradley, N., ***The Concise SGML Companion***, Addison-Wesley, 1997.

[BRE99] Brenton, C., ***Mastering Network Security,*** Sybex, Inc., 1999.

[BUS96] Buschmann, E et al., ***Pattern-Oriented Software Architecture,*** Wiley, 1996.

[DAR99] Dart, S., «Containing the Web Crisis Configuration Management», ***Proc. 1st ICSE Workshop on Webengineering,*** ACM, Los Ángeles, Mayo de 1999[12].

[DIN98] Dinucci, D., M. Giudice y L. Stiles, ***Elements of WebDesign: The Designer's Guide to a New Medium,*** 2.ª ed., Peachpit Press, 1998.

[GAM95] Gamma, E. et al., *DesignPatterns*, Addison-Wesley, 1998.

[GNA99] Gnaho, C., y F. Larcher, «A User Centered Methodology for Complex and Customimizable Web Applications engineering», ***Proc. 1st ICSE Workshop on Web engineering,*** ACM, Los Ángeles, Mayo de 1999.

[HAN99] Hansen, S., Y. Deshpande y S. Murgusan, «A Skills Hierarchy for Web Information system Development», ***Proc. 1st ICSE Workshop on WebEngineering,*** ACM, Los Ángeles, Mayo de 1999.

[ISA95] Isakowitz, T. et al., «RMM: A Methodology for Structured Hypermedia Design», ***CACM,*** vol. ***38,*** n.º 8, Agosto de 1995, pp. 43-44.

[KAE99] Kaeo, M., ***Designing Network Security,*** Cisco Press, 1999.

[LOW99] Lowe, D., «Web Engineering or Web Gardening?», ***WebNet Journal,*** vol. 1, n.º 2, Enero-Marzo de 1999.

[LYN99] Lynch, P.J., y S. Horton, ***Web Style Guide: Basic Design Principles for Creating WebSites,*** Yale University Press. 1999.

[MUR99] Murugesan, S., WebE Home Page, http://fistserv.macarthur.uws.edu.au./san/WebHome, Julio de 1999.

[NAN98] Nanard, M., y P. Kahn, «Pushing Reuse in Hypermedia Design: Golden Rules, Design pattems and constructive Templates», ***Proc. 9th ACM conf. On Hypertext and Hypermedia,*** ACM Press, 1998, pp. 11-20.

[NIE96] Nielsen, J., y A. Wagner, «User Interface Design for the WWW», ***Proc. CHI' 96 conference on Human Factors in Computing systems,*** ACM, Press, 1996, pp. 330-331.

[NOR99] Norton, **K.,** «Applying Cross Functional Methodologies to Web Development», ***Proc. 1st ICSE Workshop on WebEngineering,*** ACM, Los Ángeles, Mayo de 1995.

[OSL99] Olsina, L. et al., «Specifying Quality Characteristics and Attributes for Web Sites», ***Proc. 1st Workshop on WebEngineering,*** ACM, Los Ángeles, Mayo de 1995.

[PAR99] Pardi, W.J., ***XML in Action***, Microsoft Press, 1999.

[POW98] Powell, T.A., ***WebSite engineering,*** Prentice-Hall, 1998.

[PRE98] Pressman, R.S. (moderador), «Can Intemet-Based Applications be Engineered?», ***IEEE Software***, Septiembre de 1998, pp. 104-110.

[ROS98] Rosenfeld, L., y P. Morville, ***Information Architecture for the World Wide Web,*** O'Reilly & Associates, 1998.

[SAN96] Sano, D., ***Designing Large-Scale Web Sites: A Visual Design Methodology,*** Wiley 1996.

[SCH96] Schwabe, D., G. Rossi y S. Barbosa, «Systematic Hypermedia Application Design with OOHDM», ***Proc. Hypertext ' 96,*** pp. 116-128.

[SPO98] Spool, J.M., et al., ***Web Site Usability: A Designer's Guide,*** Morgan Kaufmann Publishers, 1998.

[STL99] St. Laurent, S., y E. Cerami, ***Building XML Applications,*** McGraw-Hill, 1999.

[TIL99] Tilley, S., y S. Huang, «On the emergence of the Renaissance Software engineer», *Proc.* ***1st ICSE Workshop on the WebEngineering,*** ACM, **Los** Ángeles, Mayo de 1999.

[12] Los procedimientos del Primer Taller ICSE sobre Ingeniería del Software se publican en la red en **http://ñstserv.marcarthur.uws.edu.au/san/icse99-WebE/ICSE99-Web-E-Proc/default.htm**

PROBLEMAS Y PUNTOS A CONSIDERAR

29.1. ¿Existen otros atributos genéricos que diferencien a las WebApps de otras aplicaciones de software? Enumere 2 ó 3.

29.2. ¿Cómo se juzga la calidad de un sitio Web? Confeccione una lista de 10 atributos de calidad prioritarios que considere más importantes.

29.3. Realice una pequeña investigación y realice un trabajo de 2 ó 3 páginas que resuma una de las tres tecnologías que se destacaron en la Sección 29.1.2.

29.4. Utilizando un sitio Web real como ejemplo, ilustre las diferentes manifestaciones del «contenido» de la WebApp.

29.5. Responda a las tres preguntas de formulación para un sitio Web con el que esté familiarizado. Desarrolle una afirmación de ámbito para el sitio Web.

29.6. Desarrolle un conjunto de perfiles para HogarSeguroInc.com o un sitio Web asignado por su profesor.

29.7. Desarrolle una lista completa de metas informativas y aplicables para HogarSeguroInc.com o un sitio Web asignado por su profesor.

29.8. Elabore un conjunto de casos de uso para HogarSeguroInc.com o un sitio Web asignado por su profesor.

29.9. ¿En qué difiere el análisis del contenido del análisis funcional y de interacción?

29.10. Realice un análisis del contenido para HogarSeguroInc.com o para un sitio Web asignado por su profesor.

29.11. Sugiera tres «reglas de oro» que servirán como guía en el diseño de WebApps.

29.12. ¿En qué difiere una configuración de diseño WebApp de una plantilla?

29.13. Seleccione un sitio Web con el que esté familiarizado y desarrolle un diseño arquitectónico razonablemente completo para el sitio. ¿Qué estructura arquitectónica seleccionó el diseñador del sitio?

29.14. Haga una investigación breve y escriba *2* ó 3 páginas que resuman el trabajo actual en las configuraciones de diseño de las WebApps.

29.15. Desarrolle un diseño arquitectónico para HogarSeguroInc.com o para un sitio Web asignado por **su** instructor.

29.16. Utilizando un sitio Web real como ejemplo, desarrolle una crítica de su interfaz de usuario y haga una recomendación que lo mejore.

29.17. Describa la manera en que una gestión de proyecto para sistemas y aplicaciones basados en Web difieren de la gestión de proyecto para el software convencional. ¿En qué se asemeja?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Durante los últimos años se han publicado cientos de libros que tratan uno o más temas de ingeniería de Web, aunque relativamente pocos tratan todos los aspectos. Lowe y Hall *(Hypertext and the Web: An Engineering Approach,* Wiley, 1999), y Powell [POW98] abarcan razonablemente estos temas. Norris, West y Warson *(Media Engineering:* **A** *Quide to Developing Information Products,* Wiley, 1997), Navarro y Khan *(Effective Web Design: Master the Essentials,* Sybex, 1998), Fleming y Koman *(Web Navigation: Designing the User Experience,* O'Reilly & Associates, 1998), y Sano [SAN96] también abarcan aspectos importantes del proceso de ingeniería.

Además de [LYN99], [DIN98] **y** [SPO98], los siguientes libros proporcionan una guía útil para los aspectos del diseño de WebApps tanto técnicos como no:

Baumgardt, M., *Creative Web Design: Tips and Tricks Step by Step,* Springer Verlag, 1998.

Donnelly, D. et al., *Cutting Edge Design: The Next Generation,* Rockport Publishing, 1998.

Holzschlag, M.E., *Web by Design: The Complete Guide,* Sybex, 1998.

Niederst, J., y R. Koman, *Web Design in a Nutshell:* **A** *Desktop Quick Reference,* O'Reilly & Associates, 1998.

Weinman, L., y R. Prirouz, *Click Here: Web Communication Design,* New Riders Publishing, 1997.

Menasce y Almeida (*Capacity Planning for Web Performance: Metrics, Models, and Methods,* Prentice-Hall, 1998) tratan la valoración cuantitativa del rendimiento de las WepApps. Amoroso *(Intrusion Detection: An Introduction to Internet Surveillance, Correlation, Trace Back, Traps, and Response,* Intrusion.Net Books, 1999) proporciona una guía detallada para los ingenieros de Web que están especializados en temas de seguridad. Umar *(Application (Re)Engineering: Building Web-Based Applications and Dealing With Legacies,* Prentice-Hall, 1997) estudia estrategias para la transformación (reingeniería) de sistemas anteriores en aplicaciones basadas en Web. Mosley *(Client Sewer Software Testing on the Desk Top and the Web,* Prentice-Hall, 1999) ha escrito uno de los mejores libros que estudian los temas de comprobación asociados con WebApps.

Haggard (*Survival Guide to Web Site Developrnent,* Microsoft Press, 1998) y *Siegel (Secrets of Successful Web Sites: Project Management on the World Wide Web,* Hayden Books, 1997) proporcionan una guía para el gestor que debe de controlar y hacer un seguimiento del desarrollo de WebApps.

Una amplia variedad de fuentes de información sobre ingeniería de Web está disponible en Internet. Una lista amplia y actualizada de referencias en sitios (páginas) web se puede obtener en **http:/www.pressman5.com.**

CAPÍTULO 30 REINGENIERÍA

EN un artículo de gran importancia escrito para la *Harvard Business Review*, Michael Hammer [HAM90] sentaba las bases para una revolución del pensamiento administrativo acerca de los procesos y computación de los negocios:

> Ha llegado el momento de dejar de pavimentar los senderos para vacas. En lugar de incrustar unos procesos anticuados en silicio y en software, deberíamos eliminarlos y volver a empezar. Tenemos que rehacer la ingeniería de nuestros negocios: hay que utilizar la potencia de la tecnología de la información moderna para rediseñar de forma radical nuestros procesos de negocios, y lograr así unas mejoras drásticas de su rendimiento.
>
> Todas las compañías funcionan de acuerdo con una gran cantidad de reglas no escritas... La reingeniería intenta apartarse de las viejas reglas acerca de la forma en que se organizan y desarrollan nuestros negocios.

Al igual que todas las revoluciones, la llamada a las armas de Hammer ha dado lugar a cambios tanto positivos como negativos. Algunas compañías han hecho un esfuerzo legítimo en rehacer su ingeniería, y los resultados han mejorado su competitividad. Otras se han basado únicamente en reducir las dimensiones y hacer subcontratas (en lugar de rehacer su reingeniería) para mejorar sus beneficios. Y como resultado se obtuvieron organizaciones reducidas con pocas probabilidades de crecer en un futuro [DEM95].

Durante la primera década del siglo veintiuno, el bombo publicitario asociado a la reingeniería ha decaído, pero el proceso en sí continúa en compañías grandes y pequeñas. La unión de la reingeniería con la ingeniería del software se encuentra en una «revisión del sistema».

VISTAZO RÁPIDO

¿Qué es? Tenga en consideración cualquier producto de tecnología que haya adquirido. Lo ve con regularidad, pero está envejeciendo. Se rompe con frecuencia, tarda en repararse y ya no representa la última tecnología. ¿Qué se puede hacer? Si el producto es de hardware, probablemente lo tirará y se comprará uno nuevo. Pero si es un software personalizado, no dispondrá la opción de tirarlo. Necesitará reconstruirlo. Creará un producto con una funcionalidad nueva, un mejor rendimiento y fiabilidad, y un mantenimiento mejorado. Eso es lo que llamamos reingeniería.

¿Quién lo hace? A nivel de negocio, la reingeniería es ejercida por especialistas de negocio (frecuentemente empresas de consultoría). A nivel de software, la reingeniería es ejecutada por ingenieros del software.

¿Por qué es importante? Vivimos en un mundo en constante cambio. Las demandas de funciones de negocios y de tecnología de información que las soportan están cambiando a un ritmo que impone mucha presión competitiva en todas las organizaciones comerciales. Tanto los negocios como el software que soportan (o es) el negocio deberán diseñarse una vez más para mantener el ritmo.

¿Cuáles son los pasos? La reingenieria de procesos de negocios (RPN) define las metas comerciales, identifica y evalúa los procesos de negocios existentes, y crea procesos comerciales revisados que mejoran las metas actuales. El proceso de reingeniería del software acompaña el análisis de inventarios, la estructuración de documentoc. la ingeniería inversa, la reestructuración de datos y la ingeniería avanzada. El objetivo de estas actividades es crear versiones nuevas de los programas existentes que exhiben una mantenibilidad de mayor calidad.

¿Cuál es el producto obtenido? Son varios los productos que se elaboran (por ejemplo, modelos análisis, modelos de diseño, procedimientos de pruebas). El resultado final es un proceso de reingeniería de negocios y/o el software de reingeniería que lo soporta.

¿Cómo puedo esta *seguro* de que lo he hecho correctamente? Utilizando las mismas prácticas que se aplican en todos los procesos de ingeniería del software —las revisiones técnicas formales evalúan los modelos de análisis y de diseños, las revisiones especializadas tienen en consideración la capacidad de aplicación comercial y la compatibilidad, y la comprobación se aplica para descubrir los errores en el contenido, en la funcionalidad y en la interoperabilidad.

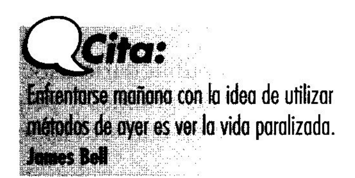
El software suele ser la realización de las reglas de negocio que describe Hammer. A medida que las reglas cambian, el software también tiene que cambiar. En la actualidad, existen compañías importantes que poseen decenas de miles de programas de computadora que prestan apoyo a las «viejas reglas del negocio». Cuando los administradores trabajan para modificar estas reglas y lograr así una mayor efectividad y competitividad, el software seguirá el mismo ritmo. En algunos casos, esto implica la creación de sistemas nuevos e importantes basados en computadora[1]. Pero en muchos otros, esto implica la modificación y/o reconstrucción de las aplicaciones existentes.

[1] Los sistemas y aplicaciones basados en Web estudiados en el Capítulo 29 son ejemplos contemporáneos.

En este capítulo se examina la reingeniería de forma descendente, comenzando por una visión general de la reingeniería de procesos de negocio y pasando entonces a una discusión más detallada de las actividades técnicas que se producen cuando se efectúa una reingeniería del software.

30.1 REINGENIERÍA DE PROCESOS DE NEGOCIO

La *Reingeniería de procesos de negocio, RPN (Business Process Reingineering,* BPR[2]) va más allá del ámbito de las tecnologías de la información y de la ingeniería del software. Entre las muchas definiciones (la mayoría de ellas, algo abstractas) que se han sugerido para la RPN, se cuenta con una publicada en la revista Fortune [STE93]: «. . .la búsqueda e implementación de cambios radicales en el proceso de negocios para lograr un avance significativo». Ahora bien ¿cómo se efectúa la búsqueda, y cómo se lleva a cabo la implementación? O lo que es más importante, ¿cómo podemos estar seguros de que el «cambio radical» que se sugiere va a dar lugar realmente a un «avance significativo» en lugar de conducimos a un caos organizativo?



30.1.1. Procesos de negocios

Un *proceso de negocio* es «un conjunto de tareas lógicamente relacionadas que se llevan a cabo para obtener un determinado resultado de negocio» [DAV90]. Dentro del proceso de negocio, se combinan las personas, los equipos, los recursos materiales y los procedimientos de negocio con objeto de producir un resultado concreto. Entre los ejemplos de negocio se incluyen el diseño de un nuevo producto, la adquisición de servicios y suministros, la contratación de nuevos empleados o el pago a proveedores. Cada una requiere un conjunto de tareas y se basa en diversos recursos dentro del negocio.

Cada proceso de negocio posee un cliente bien definido —una persona o grupo que recibe el resultado (por ejemplo: una idea, un informe, un diseño, un producto)—. Además, los procesos de negocio cruzan los límites organizativos. Requieren que distintos grupos de la organización participen en las «tareas lógicamente relacionadas» que definen el proceso.

En el Capítulo 10 se indicaba que todo sistema es en realidad una jerarquía de subsistemas. El negocio global se puede segmentar de la siguiente manera:

El negocio
  sistemas de negocio
    proceso de negocio
      subprocesos de negocio



Como ingeniero del ***software,*** *el* ***trabajo*** *de reingeniería tiene* ***lugar*** *en la parte inferior de la* ***jerarquía.*** *Sin embargo,* ***asegúrese*** *de que alguien haya pensado seriamente en el nivel anterior. Si no lo* ***han*** *hecho, su* ***trabajo*** *corre* ***algún*** *riesgo.*

Cada uno de los sistemas de negocios (también llamados función *de negocios)* están compuestos por uno o más procesos de negocio, y cada proceso de negocio está definido por un conjunto de subprocesos.

La RPN se puede aplicar a cualquier nivel de la jerarquía, pero a medida que se amplía el ámbito de la RPN (esto es, a medida que se asciende dentro de la jerarquía) los riesgos asociados a la RPN crecen de forma dramática. Por esta razón, la mayor parte de los esfuerzos de la RPN se centran en procesos o subprocesos individuales.

30.1.2. Principios de reingeniería de procesos de negocios

En muchos aspectos, la RPN tiene un objetivo y un ámbito idéntico al proceso de la ingeniería de la información (Capítulo 10). Lo ideal sería que la RPN se produjera de forma descendente, comenzando por la identificación de los objetivos principales del negocio, y culminando con una especificación mucho más detallada de las tareas que definen un proceso específico de negocios.

Hammer [HAM90] sugiere una serie de principios que nos guiarán por las actividades de la RPN cuando se comienza en el nivel superior (de negocios):

Organización en torno a los resultados, no en torno a las tareas. **Hay muchas compañías que poseen actividades de negocio compartimentadas, de tal modo que no existe una Única persona (u organización) que tenga la responsabilidad (o el control) de un cierto resultado de negocio. En tales casos, resulta difícil determinar el estado del trabajo e incluso más difícil depurar los problemas de proceso cuando esto sucede. La RPN deberá diseñar procesos que eviten este problema.**

Hay que hacer que quienes utilicen la salida del proceso lleven a cabo el proceso. **El objetivo de esta recomendación es permitir que quienes necesiten las salidas del negocio controlen todas las variables que les permitan obtener esa salida de forma temporalmente adecuada. Cuanto menor sea el número de personas distintas implicadas en el proceso, más fácil será el camino hacia un resultado rápido.**



Para encontrar una extensa información sobre la reingeniería de procesos de negocios visite la dirección **www.brint.com/BPR.htm**

[2] **BPR, en castellano RPN**

Hay que incorporar el trabajo de procesamiento de información al trabajo real que produce la información pura. A medida que la TI se distribuye, es posible localizar la mayor parte del procesamiento de información en el seno de la organización que produce **los** datos. Esto localiza el control, reduce el tiempo de comunicación y la potencia de computación se pone en manos de quienes tienen fuertes intereses en la información producida.

Hay que manipular recursos geográficamente dispersos como si estuviesen centralizados. Las comunicaciones basadas en computadoras se han sofisticado tanto que es posible situar grupos geográficamente dispersos en una misma «oficina virtual». Por ejemplo, en lugar de emplear tres turnos de ingeniería en una única localización, toda la compañía podrá tener un turno en Europa, un segundo turno en Norteaménca y un tercer turno en Asia. En todos los casos, los ingenieros trabajarán durante el día y se comunicarán empleando redes de un elevado ancho de banda.

Cita:

Tan pronto como vemos la existencia de algo viejo en algo nuevo, nos tranquilizamos.

Friedrisch Wilhelm Nietzsche

Hay que enlazar las actividades paralelas en lugar de integrar sus resultados. Cuando se utilizan diferentes grupos de empleados para realizar tareas en paralelo, es esencial diseñar un proceso que exija una continuación en la comunicación y coordinación. En caso contrario, es seguro que se producirán problemas de integración.

Hay que poner el punto de decisión en el lugar donde se efectúa el trabajo, e incorporar el control al proceso. Dentro de la jerga del diseño del software, esto sugiere una estructura organizativa más uniforme y con menos factorización.

Hay que capturar los datos una sola vez, en el lugar donde se producen. Los datos se deberán almacenar en computadoras, de tal modo que una vez recopilados no sea necesario volver a introducirlos nunca.

Todos y cada uno de los principios anteriores representan una visión «totalmente general» de la RPN. Una vez informados por estos principios, los planificadores de negocios y los diseñadores de procesos deberán empezar a procesar el nuevo diseño. En la sección siguiente, se examinará el proceso de RPN más detalladamente.

30.1.3. Un modelo de RPN

Al igual que la mayoría de las actividades de ingeniena, la reingeniería de procesos de negocio es iterativa. Los objetivos de negocio, **y** los procesos que los logran, deberán adaptarse a un entorno de negocio cambiante. Por esta razón, no existe ni principio ni fin en la RPN —se trata de un proceso evolutivo—. En la Figura 30.1 *se* representa un modelo de reingeniería de procesos de negocio. Este modelo define seis actividades:

Definición del negocio. Los objetivos de negocios se identifican en un contexto de cuatro controladores principales: *reducción de costes, reducción de tiempos, mejora de calidad y desarrollo y potenciación del personal.* Los objetivos se pueden definir en el nivel de negocios o para un componente específico del negocio.

Identificación de procesos. En esta actividad se identifican los procesos críticos para alcanzar los objetivos definidos en la definición del negocio, A continuación, pueden recibir prioridades en función de su importancia, necesidad de cambio, o cualquier otra forma que resulte adecuada para la actividad de reingeniería.

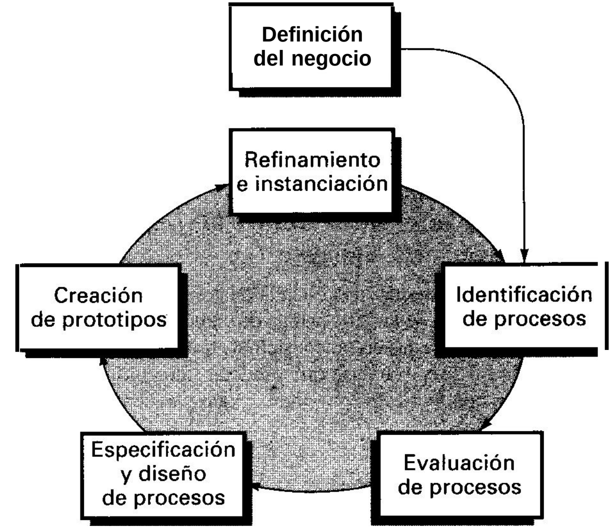


FIGURA 30.1. Un modelo de BPR.

Evaluación de procesos. Los procesos existentes deberán analizarse **y** medirse exhaustivamente. Las tareas de procesos se identificarán; los costes y los tiempos consumidos por las tareas de proceso se anotarán cuidadosamente, y se aislarán los problemas de calidad **y** rendimiento.

Especificación y diseño de procesos. Basándose en la información obtenida durante las tres primeras actividades de la RPN, se prepararán casos prácticos (Capítulo 11) para cada uno de los procesos que se tengan que rediseñar. Dentro del contexto de la RPN, los casos prácticos identifican un escenario que proporciona resultados a un cliente. Con el uso de casos prácticos como especificación del proceso, se diseña un nuevo conjunto de tareas (que se ajustan a los principios indicados en la Sección 30.2.1) para el proceso.

Creación de prototipos. Es preciso construir un prototipo del proceso de negocios rediseñado antes de integrarlo por completo en el negocio. Esta actividad «comprueba» el proceso para que sea posible efectuar refinamientos.

Refinamiento e instanciación. Basándose en la realimentación procedente del prototipo, se refina el proceso de negocio **y** después se instancia en el seno de un sistema de negocio.

En algunas ocasiones las actividades de RPN descritas anteriormente se utilizan junto con herramientas de análisis del flujo de trabajo. El objetivo de estas herramientas es construir un modelo del flujo de trabajo existente, en un esfuerzo por analizar mejor los procesos existentes. Además, para implementar las cuatro primeras actividades descritas en el modelo de procesos se pueden utilizar las

técnicas de modelado que se asocian normalmente a las actividades de ingeniería de la información tales como la planificación de estrategias de información y el análisis de áreas de negocios (Capítulo 10).

30.1.4. Advertencias

Es muy frecuente que se exagere la importancia de un nuevo enfoque de negocio - e n este caso, la RPN— como si fuese la panacea, para después criticarla con tanta severidad que pase a ser un desecho. A lo largo de los Últimos años, se ha debatido de forma exagerada acerca de la eficacia de la RPN (por ejemplo: [BLE93] y [DIC95]). En un resumen excelente del caso a favor y en contra de la RPN, Weisz (WEI95) expone su argumento de la manera siguiente:

> Resulta tentador atacar a la RPN como si se tratase de otra reencarnación de la famosa bala de plata. Desde varios puntos de vista —pensamiento de sistemas, tratamiento de personal, simple historia— habría que predecir unos índices de fallos elevados para el concepto, índices que parecen ser confirmados por la evidencia empírica. Para muchas compañías parece que la bala de plata no da en el blanco.
>
> Para otras, sin embargo, el nuevo esfuerzo de la reingeniería ha tenido evidentemente **su** fruto...

La RPN puede funcionar, si es aplicada por personas motivadas y formadas, que reconozcan que el proceso de reingeniería es una actividad continua. Si la RPN se lleva a cabo de forma efectiva, los sistemas de información se integran mejor con los procesos de negocios. Dentro del contexto de una estrategia más amplia de negocios se puede examinar la reingeniería de aplicaciones más antiguas, y también se pueden establecer de forma inteligente las prioridades de reingeniería del software.

Aunque la reingeniería de negocio sea una estrategia rechazada por una compañía, la reingeniería del software es algo que *debe* hacerse. Existen decenas de millares de sistemas heredados —aplicaciones cruciales para el éxito de negocios grandes y pequeños— que se ven afectados por una enorme necesidad de ser reconstruidos o rehechos en su totalidad.

30.2 REINGENIERÍA DEL SOFTWARE

Este escenario resulta sumamente conocido: Una aplicación ha dado servicio y ha cubierto las necesidades del negocio de una compañía durante diez o quince años. A lo largo de este tiempo, ha sido corregida, adaptada y mejorada muchas veces. Las personas se dedicaban a esta tarea con la mejor de sus intenciones, pero las prácticas de ingeniería del software buenas siempre se echaban a un lado (por la urgencia de otros problemas). Ahora la aplicación se ha vuelto inestable. Sigue funcionando, pero cada vez que intenta efectuar un cambio se producen efectos colaterales graves e inesperados. ¿Qué se puede hacer?

La imposibilidad de mantener el software no es un problema nuevo. De hecho, el gran interés por la reingeniería del software ha sido generado por un «iceberg» de mantenimiento de software que lleva creciendo desde hace más de treinta años.



SEI ofrece una variedad de recursos de reingeniería del software en la dirección: **www.sei.cmu.edu/reengineering/**

30.2.1. Mantenimiento del software

Hace casi treinta años, el mantenimiento del software se caracterizaba [CAN72] por ser como un «iceberg». Esperábamos que lo que era inmediatamente visible fuera de verdad lo que había, pero sabíamos que una enorme masa de posibles problemas y costes yacía por debajo de la superficie. A principios de los años 70, el iceberg de mantenimiento era lo suficientemente grande como para hundir un portaaviones. En la actualidad podría hundir toda la Armada.

El mantenimiento del software existente puede dar cuenta de más del 60 por 100 de las inversiones efectuadas por una organización de desarrollo, y ese porcentaje sigue ascendiendo a medida que se produce más software [HAN93]. Los lectores que tengan menos conocimientos en estos temas podrían preguntarse por qué se necesita tanto mantenimiento, y por qué se invierte tanto esfuerzo. Osborne y Chifosky [OSB90] ofrecen una respuesta parcial:

> Gran parte del software del que dependemos en la actualidad tiene por término medio entre diez y quince años de antigüedad. **Aun** cuando estos programas se crearon empleando las mejores técnicas de diseño y codificación conocidas en su época (y la mayoría no lo fueron), se crearon cuando el tamaño de **los** programas y el espacio de almacenamiento eran las preocupaciones principales. **A** continuación, se trasladaron a las nuevas plataformas, se ajustaron para adecuarlos a cambios de máquina y de sistemas operativos y se mejoraron para satisfacer nuevas necesidades del usuario; y todo esto se hizo sin tener en cuenta la arquitectura global.
>
> **El** resultado **son** unas estructuras muy mal diseñadas, una mala codificación, una lógica inadecuada, y una escasa documentación de **los** sistemas de software que ahora nos piden que mantengamos en marcha...

La naturaleza ubicua del cambio subyace en todos los tipos de trabajo del software. El cambio es algo inevitable cuando se construyen sistemas basados en computadoras; por tanto debemos desarrollar mecanismos para evaluar, controlar y realizar modificaciones.

Cita:
La capacidad de mantenimiento y de comprensión son conceptos paralelos: cuanto más difícil sea de aprender el programa, más difícil será de mantener.
Gerald Berns

Al leer los párrafos anteriores, un lector podría argumentar: «...pero yo no invierto el 60 por 100 de mi tiempo corrigiendo errores de los programas que desarrollo». Por supuesto, el mantenimiento del software es algo que va mucho más allá de «corregir errores». El mantenimiento se puede definir describiendo las cuatro actividades [SWA76] que se emprenden cuando se publica un programa para su utilización. En el Capítulo 2 se definieron cuatro actividades diferentes de mantenimiento: *mantenimiento correctivo, mantenimiento adaptativo, mejoras o mantenimiento de perfeccionamiento y mantenimiento preventivo o reingeniería.* Tan sólo el 20 por 100 de nuestros esfuerzos de mantenimiento se invertirán «corrigiendo errores». El 80 por 100 se dedicará a adaptar los sistemas existentes a los cambios de su entorno externo, a efectuar las mejoras solicitadas por los usuarios y a rehacer la ingeniería de las aplicaciones para **su** posterior utilización. Cuando se considera que el mantenimiento abarca todas estas actividades, es fácil ver por qué absorbe tanto esfuerzo.

30.2.2. Un modelo de procesos de reingeniería del software

La reingeniería requiere tiempo; conlleva un coste de dinero enorme y absorbe recursos que de otro modo podrían emplearse en preocupaciones más inmediatas. Por todas estas razones, la reingeniería no se lleva a cabo en unos pocos meses, ni siquiera en unos pocos años. La reingeniería de sistemas de información es una actividad que absorberá recursos de las tecnologías de la información durante muchos años. Esta es la razón por la cual toda organización necesita una estrategia pragmática para la reingeniería del software.

Una estrategia de trabajo también acompaña al modelo de procesos de reingeniería. Más adelante, en esta misma sección, se describirá este modelo, pero veamos en primer lugar algunos de los principios básicos.

La reingeniería es una tarea de reconstrucción, y se podrá comprender mejor la reingeniería de sistemas de información si tomamos en consideración una actividad análoga: la reconstrucción de una casa. Consideremos la situación siguiente:

Suponga que ha adquirido una casa en otro lugar. Nunca ha llegado a ver la finca realmente, pero la consiguió por un precio sorprendentemente reducido, advirtiéndosele que quizá fuera preciso reconstruirla en **su** totalidad. ¿Cómo se las arreglaría?

- Antes de empezar a construir, sería razonable inspeccionar la casa. Para determinar si necesita una reconstrucción, usted (o un inspector profesional) creará una lista de criterios para que la inspección sea sistemática.
- Antes de derribar y de construir toda la casa, asegúrese de que la estructura está en mal estado. Si la casa tiene una buena estructura, quizá sea posible remodelarla sin reconstruirla (con un coste muy inferior y en mucho menos tiempo).
- Antes de empezar a reconstruir, asegúrese de que entiende la forma en que se construyó el original. Eche una ojeada por detrás de las paredes. Comprenda el cableado, la fontanería y los detalles internos de la estructura. Aunque vaya a eliminarlos todos, la idea que haya adquirido de ellos le servirán de mucho cuando empiece a construirla.
- Si empieza a reconstruir, utilice tan solo los materiales más modernos y de mayor duración. Quizá ahora le cuesten un poquito más, pero le ayudarán a evitar un mantenimiento costoso y lento en fecha posterior.
- Si ha decidido reconstruir, tenga una actitud disciplinada. Utilice prácticas que den como resultado una gran calidad —tanto hoy como en el futuro—.



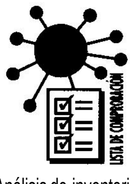
*las pasas para una casa **indicados aquí son** obvios. Asegúrese de que su consideración sobre **el software** anticuada aplica pasos que sean **tan** obvios. Piénselo. Hay mucha dinero en juego.*

Aunque los principios anteriores se centran en la reconstrucción de una casa, son aplicables igualmente a la reingeniería de sistemas y aplicaciones basados en computadoras.

Para implementar estos principios, se aplica un modelo de proceso de reingeniería del software que define las seis actividades mostradas en la Figura 30.2. En algunas ocasiones, estas actividades se producen de forma secuencial y lineal, pero esto no siempre es **así.** Por ejemplo, puede ser que la ingeniería inversa (la comprensión del funcionamiento interno de un programa) tenga que producirse antes de que pueda comenzar la reestructuración de documentos.

El paradigma de la reingeniería mostrado en la figura es un modelo cíclico. Esto significa que cada una de las actividades presentadas como parte del paradigma pueden repetirse en otras ocasiones. Para un ciclo en particular, el proceso puede terminar después de cualquiera de estas actividades.

Análisis de inventario. Todas las organizaciones de software deberán disponer de un inventario de todas sus aplicaciones. El inventario puede que no sea más que una hoja de cálculo con la información que proporciona una descripción detallada (por ejemplo: tamaño, edad, importancia para el negocio) de todas las aplicaciones activas.



Análisis de inventario

Los candidatos a la reingeniería aparecen cuando se ordena esta información en función de **su** importancia para el negocio, longevidad, mantenibilidad actual y otros criterios localmente importantes. Es entonces cuando es posible asignar recursos a las aplicaciones candidatas para el trabajo de reingeniería.

Es importante destacar que el inventario deberá revisarse con regularidad. El estado de las aplicaciones (por ejemplo, la importancia con respecto al negocio) puede cambiar en función del tiempo y, como resultado, cambiarán también las prioridades para la reingeniería.

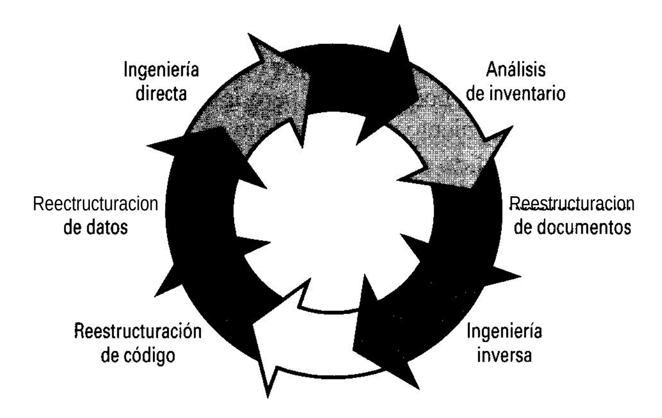


FIGURA 30.2. Un modelo de proceso de reingeniería de software.

Reestructuración de documentos. Una documentación escasa es la marca de muchos sistemas heredados. ¿Qué se puede hacer al respecto?

Opción n.º 1: La creación de documentación consume muchísimo tiempo. El sistema funciona, y ya nos apañaremos con lo que tengamos. En algunos casos, éste es el enfoque correcto. No es posible volver a crear la documentación para cientos de programas de computadoras. Si un programa es relativamente estático está llegando **al** final de vida útil, y no es probable que experimente muchos cambios: ¡dejémoslo así!

Opción n.º 2: Es preciso actualizar la documentación, pero se dispone de recursos limitados. Se utilizará un enfoque «del tipo documentar si se modifica». Quizá no se necesario volver a documentar por completo la aplicación. Más bien se documentarán por completo aquellas partes del sistema que estén experimentando cambios en ese momento. La colección de documentos Útil y relevante irá evolucionando con el tiempo.

*Opción n.º **3:*** El sistema es fundamental para el negocio, y es preciso volver a documentarlo por completo. En este caso, un enfoque inteligente consiste en reducir la documentación **al** mínimo necesario.



*Cree sólo la documentación que necesite para mejorar **su** comprensión sobre el **software** **No** para avanzar una **página** más.*

Todas y cada una de estas opciones son viables. Las organizaciones del software deberán seleccionar aquella que resulte más adecuada para cada caso.

Ingeniería inversa. El término «ingeniería inversa» tiene **sus** orígenes en el mundo del hardware. Una cierta compañía desensambla un producto de hardware competitivo en un esfuerzo por comprender los «secretos» del diseño y fabricación de su competidor. Estos secretos se podrán comprender más fácilmente si se obtuvieran las especificaciones de diseño y fabricación del mismo. Pero estos documentos son privados, y no están disponibles para la compañía que efectúa la ingeniería inversa. En esencia, una ingeniería inversa con éxito precede de una o más especificaciones de diseño y fabricación para el producto, mediante el examen de ejemplos reales de ese producto.

La ingeniería inversa del software es algo bastante similar. Sin embargo, en la mayoría de los casos, el programa del cual hay que hacer una ingeniería inversa no es el de un rival, sino, más bien, el propio trabajo de la compañía (con frecuencia efectuado hace muchos años). Los «secretos» que hay que comprender resultan incomprensibles porque nunca se llegó a desarrollar una especificación. Consiguientemente, la ingeniería inversa del software es el proceso de análisis de un programa con el fin de crear una representación de programa con un nivel de abstracción más elevado que el código fuente. La ingeniería inversa es un proceso de recuperación de diseño. Con las herramientas de la ingeniería inversa se extraerá del programa existente información del diseño arquitectónico y de proceso, e información de los datos.

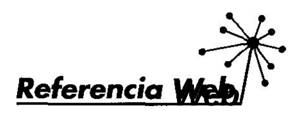
Reestructuración del código. El tipo más común de reingeniería (en realidad, la aplicación del término reingeniería sería discutible en este caso) es la reestructuración del código. Algunos sistemas heredados tienen una arquitectura de programa relativamente sólida, pero los módulos individuales han sido codificados de una forma que hace difícil comprenderlos, comprobarlos y mantenerlos. En estos casos, se puede reestructurar el código ubicado dentro de los módulos sospechosos.

Para llevar a cabo esta actividad, se analiza el código fuente mediante una herramienta de reestructuración, se indican las violaciones de las estructuras de programación estructurada, y entonces se reestructura el código (esto se puede hacer automáticamente). El código reestructurado resultante se revisa y se comprueba para asegurar que no se hayan introducido anomalías. Se actualiza la documentación interna del código.

Reestructuración de datos. Un programa que posea una estructura de datos débil será difícil de adaptar y de mejorar. De hecho, para muchas aplicaciones, la arquitectura de datos tiene más que ver con la viabilidad a largo plazo del programa que el propio código fuente.

A diferencia de la reestructuración de código, que se produce en un nivel relativamente bajo de abstracción, la estructuración de datos es una actividad de reingeniería a gran escala. En la mayoría de los casos, la reestructuración de datos comienza por una actividad de ingeniería inversa. La arquitectura de datos actual se analiza minuciosamente y se definen los modelos de

datos necesarios (Capítulo 12). Se identifican los objetos de datos y atributos y, a continuación, se revisan las estructuras de datos a efectos de calidad.



Para poder obtener una gran cantidad de recursos para la comunidad de reingeniería visite esto dirección: **www.comp.lancs.ac.uk/projects/RenaissanceWeb/**

Cuando la estructura de datos es débil (por ejemplo, actualmente se implementan archivos planos, cuando un enfoque relaciona1 simplificaría muchísimo el procesamiento), se aplica una reingeniería a los datos.

Dado que la arquitectura de datos tiene una gran influencia sobre la arquitectura del programa, y también sobre los algoritmos que lo pueblan, los cambios en datos darán lugar invariablemente a cambios o bien de arquitectura o bien de código.

Ingeniería directa (*forward engineering*). En un mundo ideal, las aplicaciones se reconstruyen utilizando un «motor de reingeniería» automatizado. En el motor se insertaría el programa viejo, que lo analizaría, reestructuraría y después regeneraría la forma de exhibir los mejores aspectos de la calidad del software. Después de un espacio de tiempo corto, es probable que llegue a aparecer este «motor», pero los fabricantes de CASE han presentado herramientas que proporcionan un subconjunto limitado de estas capacidades y que se enfrentan con dominios de aplicaciones específicos (por ejemplo, aplicaciones que han sido implementadas empleando un sistema de bases de datos específico). Lo que es más importante, estas herramientas de reingeniería cada vez son más sofisticadas.

La ingeniería directa, que se denomina también *renovación* o *reclamación* [CHI90], no solamente recupera la información de diseño de un software ya existente, sino que, además, utiliza esta información para alterar o reconstruir el sistema existente en un esfuerzo por mejorar su calidad global. En la mayoría de los casos, el software procedente de una reingeniería vuelve a implementar la funcionalidad del sistema existente, y añade además nuevas funciones y/o mejora el rendimiento global.

30.3 INGENIERÍA INVERSA

La ingeniería inversa invoca una imagen de «ranura mágica». Se inserta un listado de código no estructurado, y no documentado por la ranura, y por el otro lado sale la documentación completa del programa de computadora. Lamentablemente, la ranura mágica no existe. La ingeniería inversa puede extraer información de diseño del código fuente, pero el nivel de abstracción, la completitud de la documentación, el grado con el cual trabajan al mismo tiempo las herramientas y el analista humano, y la direccionalidad del proceso son sumamente variables [CASSS].

El *nivel de abstracción* de un proceso de ingeniería inversa y las herramientas que se utilizan para realizarlo aluden a la sofisticación de la información de diseño que se puede extraer del código fuente. El nivel de abstracción ideal deberá ser lo más alto posible. Esto es, el proceso de ingeniería inversa deberá ser capaz de derivar sus representaciones de diseño de procedimientos (con un bajo nivel de abstracción); y la información de las estructuras de datos y de programas (un nivel de abstracción ligeramente más elevado); modelos de flujo de datos y de control (un nivel de abstracción relativamente alto); y modelos de entidades y de relaciones (un elevado nivel de abstracción). A medida que crece el nivel de abstracción se proporciona al ingeniero del software información que le permitirá comprender más fácilmente estos programas.

Referencia cruzada

Lo notación indicado aquí se explica con detalle en el Capítulo 12.

La *completitud* de un proceso de ingeniería inversa alude al nivel de detalle que se proporciona en un determinado nivel de abstracción. En la mayoría de los casos, la completitud decrece a medida que aumenta el nivel de abstracción. Por ejemplo, dado un listado del código fuente, es relativamente sencillo desarrolla una representación de diseño de procedimientos completa. También se pueden derivar representaciones sencillas del flujo de datos, pero es mucho más difícil desarrollar un conjunto completo de diagramas de flujo de datos o un diagrama de transición de estados.



Se deben afrontar tres enfoques de ingeniería inversa: nivel de abstracción, completitud y direccionolidad.

La completitud mejora en proporción directa a la cantidad de análisis efectuado por la persona que está efectuando la ingeniería inversa. La *interactividad* alude al grado con el cual el ser humano se «integra» con las herramientas automatizadas para crear un proceso de ingeniería inversa efectivo. En la mayoría de los casos, a medida que crece el nivel de abstracción, la interactividad deberá incrementarse, o sino la completitud se verá reducida.

Si la *direccionalidad* del proceso de ingeniería inversa es monodireccional, toda la información extraída del código fuente se proporcionará a la ingeniería del software que podrá entonces utilizarla durante la actividad

de mantenimiento. Si la direccionabilidad es bidireccional, entonces la información se suministrará a una herramienta de reingeniería que intentará reestructurar o regenerar el viejo programa.

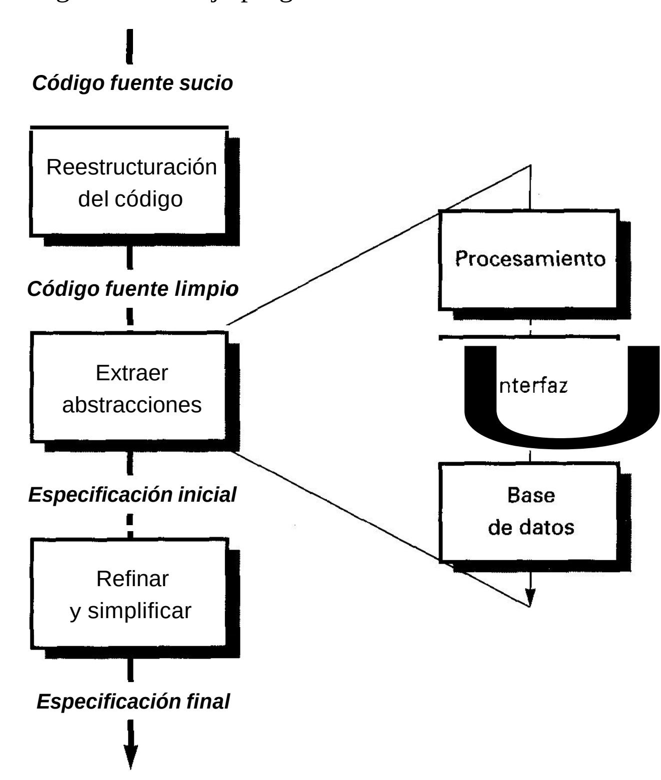


FIGURA 30.3. El proceso de ingeniería inversa.

El proceso de la ingeniena se representa en la Figura 30.3. Antes de que puedan comenzar las actividades de ingeniería inversa, el código fuente no estructurado («sucio») se *reestructura* (Sección 30.4.1) para que solamente contenga construcciones de programación estructurada[3]. Esto hace que el código fuente sea más fácil de leer, y es lo que proporciona la base para todas las actividades subsiguientes de ingeniería inversa.



La página de Recuperación de diseños y entendimiento de programas proporciona los recursos útiles para un trabajo de reingeniería: **www.sel.iit.nrc.ca/projects/dr/dr.html**

El núcleo de la ingeniería inversa es una actividad denominada *extracción de abstracciones.* El ingeniero tiene que evaluar el viejo programa y a partir del código fuente (que no suele estar documentado) tiene que extraer una especificación significativa del procesamiento que se realiza, la interfaz de usuario que se aplica, y las estructuras de datos de programa o de base de datos que se utiliza.

30.3.1. Ingeniería inversa para comprender el procesamiento

La primera actividad real de la ingeniería inversa comienza con un intento de comprender y extraer después abstracciones de procedimientos representadas por el código fuente. Para comprender las abstracciones de procedimientos, se analiza el código en distintos niveles de abstracción: sistema, programa, componente, configuración y sentencia.

La funcionalidad general de todo el sistema de aplicaciones deberá ser algo perfectamente comprendido antes de que tenga lugar un trabajo de ingeniería inversa más detallado. Esto es lo que establece un contexto para un análisis posterior, y se proporcionan ideas generales acerca de los problemas de interoperabilidad entre aplicaciones dentro del sistema. Cada uno de los programas de que consta el sistema de aplicaciones representará una abstracción funcional con un elevado nivel de detalle. También se creará un diagrama de bloques como representación de la iteración entre estas abstracciones funcionales. Cada uno de los componentes efectúa una subfunción, y representa una abstracción definida de procedimientos. En cada componente se crea una narrativa de procesamiento. En algunas situaciones ya existen especificaciones de sistema, programa y componente. Cuando ocurre tal cosa, se revisan las especificaciones para preciar si se ajustan al código existente[4].

Cita:

Existe una gran pasión por la comprensión, así como la pasión que existe por la música. Esa pasión es bastante común en los niños, pero entre la mayoría de las personas más tarde o más temprano desaparece.

Albert Einstein

Todo se complica cuando se considera el código que reside en el interior del componente. El ingeniero busca las secciones de código que representan las configuraciones genéricas de procedimientos. En casi todos los componentes, existe una sección de código que prepara los datos para su procesamiento (dentro del componente), una sección diferente de código que efectúa el procesamiento y otra sección de código que prepara los resultados del procesamiento para exportarlos de ese componente. En el interior de cada una de estas secciones, se encuentran configuraciones más pequeñas.

[3] El código se puede reestructurar automáticamente empleando un ((motor de reestructuración), —una herramienta CASE que reestructura el código fuente—.

[4] Con frecuencia, las especificaciones escritas en fases centradas de la historia del programa no llegan a actualizarse nunca. A medida que se hacen cambios, el código deja de satisfacer esas especificaciones.

Por ejemplo, suele producirse una verificación de los datos y una comprobación de los límites dentro de la sección de código que prepara los datos para su procesamiento.

Para los sistemas grandes, la ingeniería inversa suele efectuarse mediante el uso de un enfoque semiautomatizado. Las herramientas CASE se utilizan para «analizar» la semántica del código existente. La salida de este proceso se pasa entonces a unas herramientas de reestructuración y de ingeniería directa que completarán el proceso de reingeniería.

30.3.2. Ingeniería inversa para comprender los datos

La ingeniería inversa de datos suele producirse a diferentes niveles de abstracción. En el nivel de programa, es frecuente que sea preciso realizar una ingeniería inversa de las estructuras de datos internas del programa, como parte del esfuerzo general de la reingeniería. En el nivel del sistema, es frecuente que se efectúe una reingeniería de las estructuras globales de datos (por ejemplo: archivos, bases de datos) para ajustarlas a los paradigmas nuevos de gestión de bases de datos (por ejemplo, la transferencia de archivos planos a unos sistemas de bases de datos relacionales u orientados a objetos). La ingeniería inversa de las estructuras de datos globales actuales establecen el escenario para la introducción de una nueva base de datos que abarque todo el sistema.

Estructuras de datos internas. Las técnicas de ingeniería inversa para datos de programa internos se centran en la definición de clases de objetos[5]. Esto se logra examinando el código del programa en un intento de agrupar variables de programa que estén relacionadas. En muchos casos, la organización de datos en el seno del código identifica los tipos abstractos de datos. Por ejemplo, las estructuras de registros, los archivos, las listas y otras estructuras de datos que suelen proporcionar una indicación inicial de las clases.

Para la ingeniería inversa de clases, Breuer y Lano [BRE91] sugieren el enfoque siguiente:

1. Identificación de los indicadores y estructuras de datos locales dentro del programa que registran información importante acerca de las estructuras de datos globales (por ejemplo, archivos o bases de datos).
2. Definición de la relación entre indicadores y estructuras de datos locales y las estructuras de datos globales. Por ejemplo, se podrá activar un indicador cuando un archivo esté vacío; una estructura de datos local podrá servir como memoria intermedia de los cien últimos registros recogidos para una base de datos central.
3. Para toda variable (dentro de un programa) que represente una matriz o archivo, la construcción de un listado de todas las variables que tengan una relación lógica con ella.



En los próximos años los compromisos relativamente significativos en las estructuras de datos pueden conducir a tener problemas potencialmente catastróficos. Fíjese por ejemplo en el efecto del año 2000.

Estos pasos hacen posible que el ingeniero del software identifique las clases del programa que interactúan con las estructuras de datos globales.

Estructuras de bases de datos. Independientemente de su organización lógica y de su estructura física, las bases de datos permiten definir objetos de datos, y apoyan los métodos de establecer relaciones entre objetos. Por tanto, la reingeniería de un esquema de bases de datos para formar otro exige comprender los objetos ya existentes y sus relaciones.

¿Cuáles son los pasos que se pueden aplicar para aplicar la ingeniería inversa en una estructura de bases de datos existente?

Para definir el modelo de datos existente como precursor para una reingeniería que producirá un nuevo modelo de base de datos se pueden emplear los pasos siguientes [PRE94] :

1. *Construcción de un modelo de objetos inicial.* Las claves definidas como parte del modelo se podrán conseguir mediante la revisión de registros de una base de datos de archivos planos o de tablas de un esquema relacional. Los elementos de esos registros o tablas pasarán a ser atributos de una clase.
2. *Determinación de los candidatos a claves.* Los atributos se examinan para determinar si se van a utilizar o no para señalar a otro registro o tabla. Aquellos que sirvan como punteros pasarán a ser candidatos a claves.
3. *Refinamiento de las clases provisionales.* Se determina si ciertas clases similares pueden o no combinarse dentro de una Única clase.
4. *Definición de las generalizaciones.* Para determinar si se debe o no construir una jerarquía de clases con una clase de generalización como precursor de todos sus descendentes se examinan las clases que pueden tener muchos atributos similares.
5. *Descubrimiento de las asociaciones.* Mediante el uso de técnicas análogas al enfoque de CRC (Capítulo 21) se establecen las asociaciones entre clases.

[5] Para una descripción completa de estos conceptos orientados a objetos, véase la Parte Cuarta de este libro.

Una vez que se conoce la información definida en los pasos anteriores, se pueden aplicar una serie de transformaciones [PRE94] para hacer corresponder la estructura de la vieja base de datos con una nueva estructura de base de datos.

30.3.3. Ingeniería inversa de interfaces de usuario

Las IGUs sofisticadas se van volviendo de rigor para los productos basados en computadoras y para los sistemas de todo tipo. Por tanto el nuevo desarrollo de interfaces de usuario ha pasado a ser uno de los tipos más comunes de las actividades de reingeniería. Ahora bien, antes de que se pueda reconstruir una interfaz de usuario, deberá tener lugar una actividad de ingeniería inuersa.

¿Cómo puedo entender el funcionamiento de la interfaz de usuario existente?

Para comprender totalmente una interfaz de usuario ya existente (IU), es preciso especificar la estructura y comportamiento de la interfaz. Merlo y sus colaboradores [MER93] sugieren tres preguntas básicas a las cuales hay que responder cuando comienza la ingeniería inversa de la IU:

- ¿Cuáles son las acciones básicas que deberá procesar la interfaz, por ejemplo, acciones de teclado y clics de ratón?
- ¿Cuál es la descripción compacta de la respuesta de comportamiento del sistema a estas acciones?
- ¿Qué queremos decir con «sustitución», o más exactamente, qué concepto de equivalencia de interfaces es relevante en este caso?

La notación de modelado de comportamiento (Capítulo 12) puede proporcionar una forma de desarrollar las respuestas de las dos primeras preguntas indicadas anteriormente. Gran parte de la información necesaria para crear un modelo de comportamiento se puede obtener mediante la observación de la manifestación externa de la interfaz existente. Ahora bien, es preciso extraer del código la información adicional necesaria para crear el modelo de comportamiento.

Es importante indicar que una IGU de sustitución puede que no refleje la interfaz antigua de forma exacta (de hecho, puede ser totalmente diferente). Con frecuencia, merece la pena desarrollar metáforas de interacción nuevas. Por ejemplo, una solicitud de IU antigua en la que un usuario proporcione un superior (del 1 a 10) para encoger o agrandar una imagen gráfica. Es posible que una IGU diseñada utilice una barra de imágenes y un ratón para realizar la misma función.

30.4 REESTRUCTURACIÓN

La reestructuración del software modifica el código fuente y/o los datos en un intento de adecuarlo a futuros cambios. En general, la reestructuración no modifica la arquitectura global del programa. Tiende a centrarse en los detalles de diseño de módulos individuales y en estructuras de datos locales definidas dentro de los módulos. Si el esfuerzo de la reestructuración se extiende más allá de los límites de los módulos y abarca la arquitectura del software, la reestructuración pasa a ser ingeniería directa (Sección 30.5).


¿Qué beneficios se derivan de la reestructuración?

Amold [ARN89] define un cierto número de beneficios que se pueden lograr cuando se reestructura el software:

- Programas de mayor calidad —con mejor documentación y menos complejidad, y ajustados a las prácticas y estándares de la ingeniería del software moderna—.
- Reduce la frustración entre ingenieros del software que deban trabajar con el programa, mejorando por tanto la productividad y haciendo más sencillo el aprendizaje.
- Reduce el esfuerzo requerido para llevar a cabo las actividades de mantenimiento.
- Hace que el software sea más sencillo de comprobar y de depurar.

La reestructuración se produce cuando la arquitectura básica de la aplicación *es* sólida, aun cuando sus interioridades técnicas necesiten un retoque. Comienza cuando existen partes considerables del software que son Útiles todavía, y solamente existe un subconjunto de todos los módulos y datos que requieren una extensa modificación[6].

30.4.1. Reestructuración del código

La *reestructuración del código* se lleva a cabo para conseguir un diseño que produzca la misma función pero con mayor calidad que el programa original. En general, las técnicas de reestructuración del código (por ejemplo, las técnicas de simplificación lógica de Warnier [WAR74])

[6] En algunas ocasiones, resulta dificil distinguir entre una reestructuración extensa y un nuevo desarrollo. Ambos son casos de reingeniería.

modelan la lógica del programa empleando álgebra Booleana, y a continuación aplican una serie de reglas de transformación que dan lugar a una lógica reestructurada. El objetivo es tomar el código de forma de «plato de espaguetis» y derivar un diseño de procedimientos que se ajuste a la filosofía de la programación estructurada (Capítulo 16).



Aun cuando la reestructuración puede aliviar los problemas inmediatas asociadas a la depuración y los pequeños cambios, eso no es reingeniería. El beneficia real se logra salo cuando se reestructuran los datas y la arquitectura.

Se han propuesto también otras técnicas de reestructuración que puedan utilizarse con herramientas de reingeniería. Por ejemplo, Chot y Acacchi [CHO90] sugieren la creación de un diagrama de intercambio de recursos que diseña cada uno de los módulos de programa y los recursos (tipos de datos, procedimientos y variables) que se intercambian en este y otros módulos. Mediante la creación de representaciones del flujo de recursos, la arquitectura del programa se puede reestructurar para lograr el acoplamiento mínimo entre los módulos.

30.4.2. Reestructuración de los datos

Antes de que pueda comenzar la reestructuración de datos, es preciso llevar a cabo una ingeniería inversa, llamada *análisis del código fuente*. En primer lugar se evaluarán todas las sentencias del lenguaje de programación con definiciones de datos, descripciones de archivos, de E/S, y descripciones de interfaz. El objetivo es extraer elementos y objetos de datos, para obtener información acerca del flujo de datos, así como comprender las estructuras de datos ya existentes que se hayan implementado. Esta actividad a veces se denomina *análisis de datos* [RIC89].

Una vez finalizado el análisis de datos, comienza el *rediseño de datos*. En **su** forma más sencilla se emplea un paso de *estandarización de rediseño de datos* que clarifica las definiciones de datos para lograr una consistencia entre nombres de objetos de datos, o entre formatos de registros físicos en el seno de la estructura de datos o formato de archivos existentes. Otra forma de rediseño, denominada *racionalización de nombres de datos*, garantiza que todas las convenciones de denominación de datos se ajusten a los estándares locales, y que se eliminen las irregularidades a medida que los datos fluyen por el sistema.

Cuando la reestructuración sobrepasa la estandarización y la racionalización, se efectúan modificaciones físicas en las estructuras de datos ya existentes con objeto de hacer que el diseño de datos sea más efectivo. Esto puede significar una conversión de un formato de archivo a otro, o, en algunos casos, una conversión de un tipo de base de datos a otra.

30.5 INGENIERÍA DIRECTA (FORWARD ENGINEERING)

Un programa que posea flujo de control es el equivalente gráfico de un plato de espaguetis con «módulos», es decir con 2.000 sentencias de longitud; con pocas líneas de comentario útiles en 290.000 sentencias de código fuente, y **sin** ninguna otra documentación que se deba modificar para ajustarle a los requisitos de cambios del usuario. Se puede decir que disponemos de las opciones siguientes:

1. La posibilidad de esforzarse por efectuar una modificación tras otra, luchando con el diseño implícito y con el código fuente para implementar los cambios necesarios.
2. La posibilidad de intentar comprender el funcionamiento interno más amplio del programa, en un esfuerzo por hacer que las modificaciones sean más efectivas.

¿Qué opciones existen cuando nos enfrentamos con un programa de diseño e implementeciónpobres?

3. La posibilidad de rediseñar, recodificar y comprobar aquellas partes del software que requieran modificaciones, aplicando un enfoque de ingeniería del software a todos los segmentos revisados.
4. La posibilidad de rediseñar, recodificar y comprobar el programa en su totalidad, utilizando herramientas **CASE** (herramientas de reingeniería) que servirán para comprender el diseño actual.

No existe una Única opción «correcta». Las circunstancias pueden imponer la primera opción, aun cuando las otras sean las preferidas.

En lugar de esperar a que se reciba una solicitud de mantenimiento, la organización de desarrollo o de soporte utilizará los resultados del análisis de inventario para seleccionar el programa (1) que continúe utilizándose durante un número determinado de años, (2) que se esté utilizando con éxito en la actualidad, y **(3)** que tenga probabilidades de sufrir grandes modificaciones o mejoras en un futuro próximo. Entonces, se aplicarán las opciones **2, 3** ó 4 anteriores.

Este enfoque de mantenimiento preventivo fue introducido por Miller [MIL81] con el nombre de «reajuste estructurado». Definía este concepto como «la aplica-

ción de las metodologías de hoy a los sistemas de ayer para prestar apoyo a los requisitos del mañana».

A primera vista, la sugerencia de volver a desarrollar un gran programa cuando ya existe una versión operativa puede parecer más bien extravagante. Sin embargo, antes de pasar a emitir un juicio, considérense los puntos siguientes:

1. El coste de mantener una línea de código fuente puede estar entre veinte y cuarenta veces por encima del coste del desarrollo inicial de esa línea.
2. El rediseño de la arquitectura del software (del programa y/o de las estructuras de datos) empleando conceptos de diseño modernos puede facilitar mucho el mantenimiento futuro.
3. Dado que ya existe un prototipo del software, la productividad de desarrollo deberá ser mucho más elevada que la media.
4. En la actualidad, el usuario ya tiene experiencia con el software. Por tanto, los nuevos requisitos y la dirección del cambio se podrán estimarse con mucha más facilidad.
5. Las herramientas CASE para la reingeniería automatizarán algunas partes del trabajo.
6. Cuando finalice el mantenimiento preventivo, se dispondrá de una configuración completa del software (documentos, programas y datos).



La reingeniería es muy similar al hecho de lavarse los dientes. Se puede pensar en mil razones y tardar en lavárselas, dejándolo para más tarde. Pero, al final, estas tácticas de retrasar las cosas siempre volverán a rondarnos.

Cuando una organización de desarrollo de software vende el software como un producto, el mantenimiento preventivo se ve como «nuevas versiones» del programa. Un gran desarrollador de software local (por ejemplo, un grupo de desarrollo de software de sistemas para una compañía de productos de un consumidor de gran volumen) puede tener entre 500 y 2.000 programas de producción dentro de su dominio de responsabilidad. Se podrán asignar prioridades a estos programas según su importancia, y a continuación se revisarán esos programas como los candidatos para el mantenimiento preventivo.

El proceso de ingeniería del software aplica principios, conceptos y métodos de ingeniería del software para volver a crear las aplicaciones existentes. En la mayoría de los casos, la ingeniería directa no se limita a crear un equivalente moderno de un programa anterior, sino que más bien se integran los nuevos requisitos y las nuevas tecnologías 'en ese esfuerzo de volver a aplicar reingeniería. El programa que se ha vuelto a desarrollar amplía las capacidades de la aplicación anterior.

30.5.1. Ingeniería directa para arquitecturas cliente/servidor

A lo largo de la última década, muchas aplicaciones de grandes computadoras han sufrido un proceso de reingeniería para adaptarlas a arquitecturas cliente/servidor (C/S). En esencia, los recursos de computación centralizados (incluyendo el software) se distribuyen entre muchas plataformas cliente. Aunque se puede diseñar toda una gama de entornos distribuidos distintos, la aplicación típica de computadora central que sufre un proceso de reingeniería para adoptar una arquitectura cliente/servidor posee las características siguientes:

- la funcionalidad de la aplicación migra hacia todas las computadoras cliente;
- se implementan nuevas interfaces IGU en los centros clientes;
- las funciones de bases de datos se le asignan al servidor;
- la funcionalidad especializada (por ejemplo, los análisis de computación intensiva) pueden permanecer en el centro del servidor; y
- los nuevos requisitos de comunicaciones, seguridad, archivado y control deberán establecerse tanto en el centro cliente, como en el centro servidor.

Referencia cruzada

La ingeniería del software cliente/servidor se estudia en el Capítulo 28.

Es importante tener en cuenta que la migración desde computadoras centrales a un proceso C/S requiere tanto una reingeniería del negocio como una reingeniería del software. Además, es preciso establecer una «infraestructura de red de empresa». [JAY94]



En algunos casos, los sistemas C/S o los sistemas OO diseñados para reemplazar una aplicación antigua deberán enfocarse como un proyecto nuevo de desarrollo. La reingeniería entra en juego solo cuando los elementos de un sistema antiguo se van a integrar con la nueva arquitectura. En algunos casos, quizás es mejor no aceptar la vieja y crear una funcionalidad nueva e idéntica.

La reingeniería de aplicaciones C/S comienza con un análisis exhaustivo del entorno de negocios que abarca la computadora central existente. Se pueden identificar tres capas de abstracción (Fig. **30.4).** La base de datos se encuentra en los cimientos de la arquitectura cliente/servidor, y gestiona las transacciones y consultas que proceden de las aplicaciones de servidor. Sin embargo, estas transacciones y consultas tienen que ser controladas en el contexto de un conjunto de reglas de negocio (definidas por un proceso de negocio ya existente o que ha experimentado una reingeniería). Las aplicaciones de cliente proporcionan la funcionalidad deseada para la comunidad de usuarios.

Las funciones del sistema de gestión de bases de datos ya existente y la arquitectura de datos de la base de datos existente deberán sufrir un proceso de ingeniería inversa como precedente para el rediseño de la capa de fundamento de la base de datos. En algunos casos, se crea un nuevo modelo de datos (Capítulo 12). En todos los casos, la base de datos C/S sufre un proceso de reingeniería para asegurar que las transacciones se ejecutan consistentemente; para asegurar que todas las actualizaciones se efectúen únicamente por usuarios autorizados; para asegurar que se impongan las reglas de negocios fundamentales (por ejemplo, antes de borrar el registro de un proveedor, el servidor se asegura de que no exista ningún registro pendiente, contrato o comunicación para ese proveedor); para asegurar que se puedan admitir las consultas de forma eficaz; y para asegurar que se ha establecido una capacidad completa de archivado.

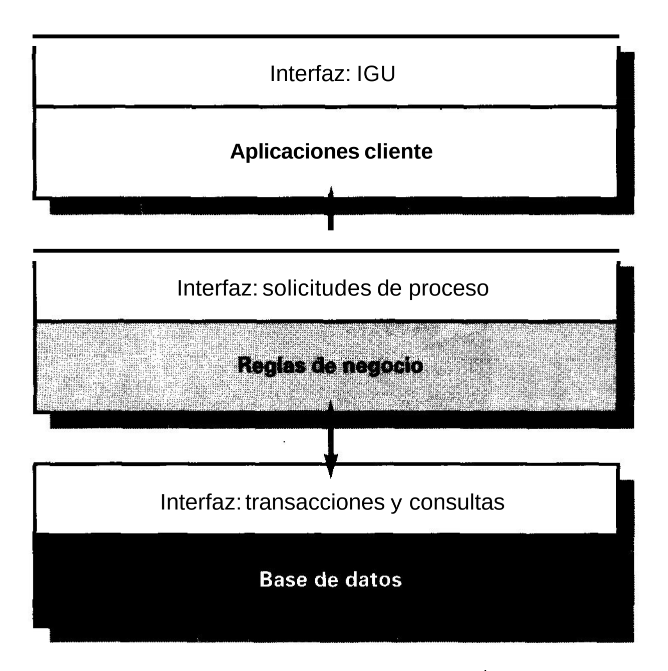


FIGURA 30.4. Proceso de reingeniería de aplicaciones de computadores centrales a cliente/servidor.

La capa de reglas de negocios representa el software residente tanto en el cliente como en el servidor. Este software lleva a cabo las tareas de control y coordinación para asegurar que las transacciones y consultas entre la aplicación cliente y la base de datos se ajusten a los procesos de negocios establecidos.

La capa de aplicación del cliente implementa las funciones de negocios requeridas por grupos específicos de usuarios finales. En muchos casos, se segmenta una aplicación de computadora central en un cierto número de aplicaciones más pequeñas, sometidas a reingeniería, y adecuadas para su funcionamiento de sobremesa. La comunicación entre aplicaciones de sobremesa (cuando es necesaria) será controlada por la capa de reglas de negocios.

Un estudio completo sobre el diseño y reingeniería de software cliente/servidor es más adecuado para otros libros especializados en este materia. El lector interesado deberá consultar [VAS93], [INM93[y [BER92].

30.5.2. Ingeniería directa para arquitecturas orientadas a objetos

La ingeniería del software orientada a objetos se ha transformado en el paradigma opcional de desarrollo para muchas organizaciones de software. Sin embargo, ¿qué sucede con las aplicaciones existentes que se desarrollaron empleando métodos convencionales? En algunos casos, la respuesta consiste en dejar estas aplicaciones tal y como eran. Pero en otros casos, es preciso aplicar una reingeniería a las viejas aplicaciones para que se puedan integrar fácilmente en grandes sistemas orientados a objetos.

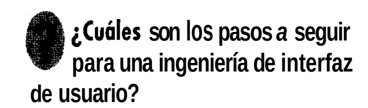
La reingeniería del software convencional para producir una implementación orientada a objetos hace uso de muchas de las mismas técnicas descritas en la Cuarta Parte de este libro. En primer lugar, se hace una ingeniería inversa del software existente para que sea posible crear los modelos adecuados de datos, funcional y de comportamiento. Si el sistema que se aplica a la reingeniería extiende la funcionalidad o comportamiento de la aplicación original, se crean casos prácticos (Capítulos 11 y 21). Los modelos de datos creados durante la ingeniería inversa se utilizan entonces junto con un modelado CRC (Capítulo 21) para establecer la base para la definición de clases. Las jerarquías de clases, los modelos de relaciones entre objetos, los modelos de comportamiento de objetos, y los subsistemas se definen a continuación, y comienza el diseño orientado a objetos.

A medida que la ingeniería directa orientada a objetos pasa del análisis hasta el diseño, se podrá invocar el modelo de proceso de ISBC (Capítulo 27). Si la aplicación existente se encuentra con un dominio ya ha sido popularizada por muchas aplicaciones orientadas a objetos, es probable que exista una biblioteca robusta de componentes y que se pueda utilizar durante la ingeniería directa.

Para aquellas clases que sea preciso construir partiendo de cero, quizá sea posible reutilizar algoritmos y estructuras de datos procedentes de la aplicación convencional ya existente. Si embargo, es preciso volver a diseñarlos para ajustarse a la arquitectura orientada a objetos.

30.5.3. Ingeniería directa para interfaces de usuario

Cuando las aplicaciones migran desde la computadora central a la computadora de sobremesa, los usuarios ya no están dispuestos a admitir unas interfaces de usuarios arcanas basadas en caracteres. De hecho, una parte significativa de todos los esfuerzos invertidos en la transición desde la computación de computadora central hasta la arquitectura cliente/servidor se pueden reinvertir en la reingeniería de las interfaces de usuario de la aplicación cliente.



Merlo y colaboradores [MER95] sugieren el modelo siguiente para la reingeniería de interfaces de usuario:

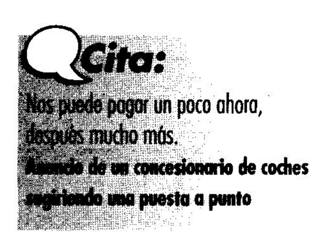
1. *Comprender la interfaz original y los datos que se trasladan entre ella y el resto de la aplicación.* El objetivo es comprender la forma en que los demás elementos del programa interactúan con el código existente que implementa la interfaz. Si se ha de desarrollar una nueva IGU, entonces el flujo de datos entre la IGU y el resto del programa deberá ser consecuente con los datos que en la actualidad fluyen entre la interfaz basada en caracteres y el programa.
2. *Remodelar el comportamiento implícito en interfaz existente para formar una serie de abstracciones que tengan sentido en el contexto de una IGU.* Aunque el modelo de interacción pueda ser radicalmente distinto, el comportamiento de negocios mostrado por los usuarios de la interfaz nueva y vieja (cuando se consideran en función del escenario de utilización) deberán seguir siendo los mismos. La interfaz rediseñada deberá seguir permitiendo que el usuario muestre el comportamiento de negocios adecuado. Por ejemplo, cuando se efectúa una consulta en una base de datos, es posible que, para especificar la consulta, la interfaz vieja necesite una serie larga de órdenes basadas en textos. La IGU sometida a reingeniería puede hacer que la consulta sea más eficiente, reduciéndola a una pequeña secuencia de selecciones con el ratón, pero la intención y el contenido de la consulta deberán permanecer intactas.
3. *Introducir mejoras que hagan que el modo de interacción sea más eficiente.* Los fallos ergonómicos de la interfaz existente se estudian y se corrigen en el diseño de la IGU nueva.
4. *Construir e integrar la IGU nueva.* La existencia de bibliotecas de clases y de herramientas de cuarta generación puede reducir el esfuerzo requerido para construir la IGU de forma significativa. Sin embargo, la integración con el software de aplicación ya existente puede llevar más tiempo. Para asegurarse de que la IGU no propague unos efectos adversos al resto de la aplicación es preciso tener cuidado.

30.6 LA ECONOMÍA DE LA REINGENIERÍA

En un mundo perfecto, todo programa que no se pudiera mantener se retiraría inmediatamente, para ser sustituido por unas aplicaciones de alta calidad, fabricadas mediante reingeniería y desarrolladas empleando las prácticas de la ingeniería del software modernas. Sin embargo, vivimos en un mundo de recursos limitados. La reingeniería consume recursos que se pueden utilizar para otros propósitos de negocio. Consiguientemente, antes de que una organización intente efectuar una reingeniería de la aplicación existente, deberá llevar a cabo un análisis de costes y beneficios.

Sneed [SNE95] ha propuesto un modelo de análisis de costes y beneficios para la reingeniería. Se definen nueve parámetros:

P_1 = coste de mantenimiento anual actual para una aplicación;
P_2 = coste de operación anual de una aplicación;
P_3 = valor de negocios anual actual de una aplicación;
P_4 = coste de mantenimiento anual predicho después de la reingeniería;
P_5 = coste de operaciones anual predicho después de la reingeniería;
P_6 = valor de negocio actual predicho después de la reingeniería;
P_7 = costes de reingeniería estimados;
P_8 = fecha estimada de reingeniería;
P_9 = factor de riesgo de la reingeniería (P_9 = 1,0 es el valor nominal);
L = vida esperada del sistema.



El coste asociado al mantenimiento continuado de una aplicación candidata (esto es, si no se realiza la reingeniería) se puede definir como:

$$C_{mant} = [P_3 - (P_1 + P_2)] \times L \qquad (30.1)$$

Los costes asociados con la reingeniería se definen empleando la relación siguiente:

$$C_{reing} = [P_6 - (P_4 + P_5)] \times (L - P_8) - (P_7 \times P_9)] \qquad (30.2)$$

Empleando los costes presentados en la ecuaciones (30.1) y (30.2), los beneficios globales de la reingeniería se pueden calcular en la forma siguiente:

$$\text{Beneficio y coste} = C_{\text{reing}} - C_{\text{mant}} \qquad (30.3)$$

El análisis de costes y beneficios presentados en las ecuaciones anteriores se puede llevar a cabo para todas aquellas aplicaciones de alta prioridad que se hayan identificado durante un análisis de inventario (Sección 30.2.2). Aquellas aplicaciones que muestren el mayor beneficio en relación con los costes podrán destinarse a la reingeniería, mientras que las demás podrán ser propuestas hasta que se disponga de más recursos.

RESUMEN

La ingeniería se produce en dos niveles distintos de abstracción. En el nivel de negocios, la reingeniería se concentra en el proceso de negocios con la intención de efectuar cambios que mejoren la competitividad en algún aspecto de los negocios. En el nivel del software, la reingeniería examina los sistemas y aplicaciones de información con la intención de reestructurarlos o reconstruirlos de tal modo que muestren una mayor calidad.

La reingeniería de procesos de negocio (RPN) define los objetivos de negocios, identifica y evalúa los procesos de negocios ya existentes (en el contexto de los objetivos definidos), especifica y diseña los procesos revisados, y construye prototipos, refina e instancia esos procesos en el seno de un negocio. La RPN tiene un objetivo que va más allá del software. Su resultado suele ser la definición de formas en que las tecnologías de la información puedan prestar un mejor apoyo a los negocios.

La reingeniería del software abarca una serie de actividades entre las que se incluye el análisis de inventario, la reestructuración de documentos, la ingeniería inversa, la reestructuración de programas y datos, y la ingeniería directa. El objetivo de esas actividades consiste en crear versiones de los programas existentes que muestren una mayor calidad, y una mejor mantenibilidad —se trata de programas que sean viables hasta bien entrado el siglo veintiuno—.

El análisis de inventarios permite que una organización estime todas y cada una de las aplicaciones sistemáticamente, con el fin de determinar cuáles son las candidatas para una reingeniería. La reestructuración de documentos crea un marco de trabajo de documentos necesario para el apoyo de una cierta aplicación a largo plazo. La ingeniería inversa es el proceso de analizar un programa en un esfuerzo por extraer información acerca de los datos, de su arquitectura y del diseño de procedimientos. Por Último, la ingeniería directa reconstruye el programa empleando prácticas de ingeniería moderna del software y la información obtenida durante la ingeniería inversa.

Los costes y beneficios de la reingeniería se pueden determinar de forma cuantitativa. El coste del status quo, esto es, los costes asociados al mantenimiento y soporte que conlleva una aplicación existente se puede comparar con los costes estimados de la reingeniería, y con la reducción resultante de los costes de mantenimiento. En casi todos los casos en que un programa tiene una vida larga y muestra en la actualidad un mantenimiento dificultoso, la reingeniería representa una estrategia de negocios eficiente en relación con los costes.

REFERENCIAS

[ARN89] Arnold, R. S., «Software Restructuring», ***Proc. IEEE,*** vol. 77, n.° 4, Abril de 1989, pp. 607-617.

[BER92] Berson, A., *Client/Server* ***Architecture,*** McGraw-Hill, 1992.

[BLE93] Bleakley, F.R., «The Best Plans: Many Companies Try Management Fads, Only to See Them Flop», *The Wall* ***Street Journal,*** 6 de Julio de 1993, p. 1.

[BRE91] Breuer, P.T., y K. Lano, «Creating Specification From Code: Reverse-Engineering Techniques», ***Journal*** *of Software* ***Muintenance: Research and Practice,*** vol. 3, Wiley, 1991, pp. 145-162.

[CAN72] Canning, R., «The Maintenance "Iceberg"», ***EDP Analyzer,*** vol. 10, n.° 10, Octubre de 1972.

[CASSS] «Case Tools for Reverse Engineering», ***CASE Outlook,*** CASE Consulting Group, Lake Oswego, OR, vol. 2, n.° 2, 1988, pp. 1-15.

[CHI90] Chifofsky, E.J., y J.H. Cross, II, «Reverse Engineering and Design Recovery: A Taxonomy», ***IEEE*** *Software,* Enero de 1990, pp. 13-17.

[DAV90] Davenport, T.H., y J.E. Young, «The New Industrial Engineering: Information Technology and Business Process Redesign», *Sloan Management* ***Review,*** Summer 1990, pp. 11-27.

[DEM95] DeMarco, T., «Lean and Mean», ***IEEE Software,*** Noviembre de 1995, pp. 101-102.

[DIC95] Dickinson, B., ***Strategic Business Reengineering,*** LCI Press, 1995.

[HAM90] Hammer, M., «Reengineer Work: Don't Automate, Obliterate», ***Harvard Business Review,*** Julio-Agosto de 1990, pp. 104-112.

[HAN93] Hanna, M., «Maintenance Burden Begging for a Remedy», ***Datamation,*** Abril de 1993, pp. 53-63.

[INM93] Inmon, W.H., ***Developing Client Server Applications,*** QED Publishing, 1993.

[JAY94] Jaychandra, Y., ***Re-engineering the Networked Enterprise***», McGraw-Hill, 1994.

[MAR94] Markosian, L. et al., «Using an Enabling Technology to Reengineer Legacy Systems, ***CACM,*** vol. 37, n.° 5, Mayo de 1994, pp. 58-70.

[MER93] Merlo, E. et al., «Reverse Engineering of User Interfaces», ***Proc. Working Conference on Reverse Engineering,*** IEEE, Baltimore, MD, Mayo de 1993, pp. 171-178.

[MER95] Merlo, E. et al., «Reengineering User Interfaces», ***IEEE Software,*** Enero de 1995, pp. 64-73.

[MIL81] Miller, J., *Techniques of Prograrn and Systern Maintenance* (G. Parikh, ed.), Winthrop Publishers, 1981.

[OSB90] Osbome, W.M., y E.J. Chifofsky, «Fitting Pieces to the Maintenance Puzzle», *IEEE Software*, Enero de 1990, pp. 10-11.

[PRE94] Premerlani, W.J., y M.R. Blaha, «An Approach for Reverse Engineering of Relational Databases», *CACM*, vol. 37, n.° 5, Mayo de 1994, pp. 42-49.

[RIC89] Ricketts, J.A., J.C. DelMonaco y M.W. Weeks, «Data Reengineering for Application Systems», *Proc. Conf. Software Maintenance-1989*, IEEE, 1989, pp. 174-179.

[STE93] Steward, T.A., «Reengineering: The Hot New Managing Tool», *Fortune*, 23 de Agosto de 1993, pp. 41-48.

[SWA76] Swanson, E.B., «The Dimensions of Maintenance», *Proc. 2nd Intl. Conf. Software Engineering*, IEEE, Octubre de 1976, pp. 492-497.

[VAS93] Vaskevitch, D., *Client/Server Strategies*, IDG Books, San Mateo, CA, 1993.

[WAR74] Wamier, J.D., *Logical Construction of Prograrns*, VanNostrand Reinhold, 1974.

[WEI95] Weisz, M., «BPR is Like Teenage Sex», *Arnerican Prograrnrner*, vol. 8, n.° 6, Junio de 1995, pp. 9-15.

PROBLEMAS Y PUNTOS A CONSIDERAR

30.1. Considere cualquier trabajo que haya desempeñado en los últimos cinco años. Describa el proceso de negocios en que haya desempeñado **su** función. Utilice el modelo RPN descrito en la Sección 30.1.3 para recomendar cambios en el proceso con la intención de hacerlo más eficiente.

30.2. Realice una investigación acerca de la eficacia de la reingeniería de procesos de negocios. Presente argumentos a favor y en contra de este enfoque.

30.3. Su profesor seleccionará uno de los programas que haya desarrollado alguien de la clase durante el curso. Intercambie **su** programa aleatoriamente con cualquier otra persona de la clase. No le explique ni revise el programa. A continuación, implemente una mejora (especificada por el instructor) en el programa que haya recibido.

a. Lleve a cabo todas las tareas de ingeniería del software incluyendo una breve revisión (pero no con el autor del programa).
b. Lleve la cuenta cuidadosamentede todos los errores encontrados durante la comprobación.
c. Describa su experiencia en clase.

30.4. Explore la lista de comprobación del análisis de inventario presentada en el sitio Web SEPA e intente desarrollar un sistema de evaluación cuantitativo del software que *se* pueda aplicar a los programas existentes en un esfuerzo de seleccionar los programas candidatos para su reingeniería. **Su** sistema deberá ir más allá del análisis económico presentado en la Sección 30.6.

30.5. Sugiera alternativas para el papel y el lápiz o para la documentación electrónica convencional que puedan servir como base para la reestructuración de documentos. [Pista: Piense en las nuevas tecnologías descriptivas que se podrían utilizar para comunicar el objetivo del software.]

30.6. Algunas personas creen que las técnicas de inteligencia artificial incrementarán el nivel de abstracción de los procesos de ingeniería inversa. Efectúe una investigación sobre este tema (esto es, el uso de la IA para ingeniería inversa) y escriba un artículo breve que se decante por este tema.

30.7. ¿Por qué es más difícil lograr la completitud a medida que el nivel de abstracción crece?

30.8. ¿Por qué tiene que incrementarse la interactividad si es preciso incrementarse la completitud?

30.9. Obtenga literatura de productos correspondientes a tres herramientas de ingeniería inversa y presente **sus** características en clase.

30.10. Existe una diferencia sutil entre la reestructuración y la ingeniería directa. ¿En qué consiste?

30.11. Investigue en la literatura para hallar uno o dos artículos que describan casos prácticos de reingeniería de grandes computadoras para producir una arquitectura cliente/servidor. Presente un resumen.

30.12. ¿Cómo se determinarían los valores de P_4 a P_7 en el modelo de costes y beneficios presentado en la Sección 30.6?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Al igual que muchos tópicos candentes dentro de la comunidad de negocios, el bombo publicitario de la reingeniería de procesos de negocios ha dado pie a una visión más pragmática del tema. Hammer y Champy *(Reengineering the Corporation,* HarperCollins, 1993) anticiparon gran interés por el tema con **su** best seller. Posteriormente, Hammer *(Beyond Reengineering: How the Processed-Centered Organization is Changing Our Work and Our Lives*, Harper-Collins, 1997) refinó su visión centrándose en temas «centrados en procesos».

Los libros de Andersen *(Business Process Irnprovernent Toolhox,* American Society for Quality, 1999), Harrington et al. *(Business Process Improvement Workhook,* McGraw-Hill, 1997), Hunt *(Process Mapping: How to Reengineer Your Business Processes,* Wiley 1996), y Carr y Johanson *(Best Practices in Reengineering: What Works and What Doesn't in the Reengineering Process,* McGraw-Hill, 1995) presentan casos prácticos y líneas generales detalladas para RPN.

Feldmann *(The Practica1 Guide to Business Process Reengineering Using IDEFO,* Dorset House, 1998) estudia una notación modelada que ayuda en la RPN. Berztiss *(Sofiare Methods for Business Reengineering,* Springer, 1996) y Spurr y colaboradores *(Software Assistance for Business Reengineering,* Wiley, 1994) estudian las herramientas y técnicas que facilitan la RPN.

Relativamente pocos libros se han dedicado a la reingeniería del software. Rada *(Reengineering Software: How to Reuse Programming to Build New, State-Of-The-Art Software*, Fitzroy Dearborn Publishers, 1999) se centra en la reingeniería a pn nivel técnico. Miller *(Reengineering Legacy Software Systems*, Digital Press, 1998) «proporciona un marco de trabajo para mantener los sistemas de aplicaciones sincronizados con las estrategias de negocios y con los cambios tecnológicos». Umar *(Application(Re)Engineering: Building Web-Based Applications and Dealing With Legacies*, Prentice Hall, 1997) proporciona una guía valiosa para aquellas organizaciones que quieren transformar los sistemas antiguos en un entorno basado en Web. Cook *(Building Enterprise Information Architectures: Reengineering Information Systems*, Prentice Hall, 1996) estudia el puente que existe entre RPN y la tecnología de información. Aiken *(Data Reverse Engineering*, McGraw-Hill, 1996) estudia cómo reclamar, reorganizar y reutilizar los datos de la organización. Arnold *(Software Reengineering*, IEEE Computer Society Press, 1993) ha publicado una antología excelente de los primeros trabajos sobre las tecnologías de reingeniería del software.

En Intemet se disponen de gran variedad fuentes de información sobre la reingeniería de procesos de negocios y la reingeniería del software. Una lista actualizada de referencias a sitios (páginas) web se puede encontrar en la dirección **http:/www.pressman5.com.**

.

CAPÍTULO

31 INGENIERÍA DEL SOFTWARE ASISTIDA POR COMPUTADORA

TODO el mundo ha oído ese proverbio que habla de los hijos del zapatero: el zapatero está tan ocupado haciendo zapatos para los demás que sus hijos no tienen sus propios zapatos. Antes de los años 90, muchos de los ingenieros de software fueron los hijos del zapatero. Aunque estos profesionales técnicos construyeron sistemas complejos y productos que automatizan los trabajos de otros, no utilizaron mucha automatización para ellos mismos.

En la actualidad, los ingenieros de software han recibido por fin su primer par de zapatos nuevos —la ingeniería del software asistida por computadora (CASE)—. Los zapatos no vienen con tanta variedad como querrían, a veces son un poco duros y en algunos casos resultan incómodos, no proporcionan una sofisticación suficiente para los que los gustan de vestir bien, y no siempre coinciden con la ropa que utilizan los que desarrollan el software. Ahora bien, suponen una prenda absolutamente esencial en el guardarropa del que desarrolla el software y, con el tiempo, serán más cómodos, se utilizarán con más facilidad y se adaptarán mejor a las necesidades de cada uno de los desarrolladores.

En los capítulos anteriores de este libro se ha intentado proporcionar una explicación razonable para comprender lo que sucede entre bastidores dentro de la tecnología de la ingeniería del software. En este capítulo, nuestro centro de interés se desplaza hacia las herramientas y entornos que ayudarán a automatizar el proceso del software.

VISTAZO RÁPIDO

¿Qué es? Las herramientas CASE ayudan a los gestores y practicantes de la ingeniería del software en todas las actividades asociadas a los procesos de software. Automatizan las actividades de gestión de proyectos, gestionan todos los productos de los trabajos elaborados a través del proceso, y ayudan a los ingenieros en el trabajo de análisis, diseño y codificación. Las herramientas CASE se pueden integrar dentro de un entorno sofisticado.

¿Quién lo hace? Los gestores de proyectos y los ingenieros del software.

¿Por qué es importante? La ingeniería del software es difícil. Las herramientas que reducen la cantidad de esfuerzo que se requiere para producir un producto de trabajo o para realizar algún hito tiene un beneficio sustancial. Pero hay algo incluso más importante. Las herramientas pueden proporcionar nuevas formas de observar la información de la ingeniería del software —formas que mejoran la perspicacia del ingeniero que realiza el trabajo—. Esto conduce a tomar mejores decisiones y conseguir una calidad mejor del software.

¿Cuáles son los pasos? CASE se utiliza junto con el modelo de procesos que se haya elegido. Si se dispone de un juego completo de herramientas, se utilizará CASE a lo largo de casi todos los pasos del proceso de software.

¿Cuál es el producto obtenido? Las herramientas CASE ayudan al ingeniero del software en la producción de resultados de alta calidad. Además, disponer de automatización permite que el usuario de CASE elabore resultados adicionales y personalizados que no serán fáciles ni prácticos de producir sin el soporte de las herramientas.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Utilice las herramientas como complemento de las prácticas de ingeniería del software —no como sustitutivo—. Antes de poder utilizar las herramientas eficazmente, deberán aprenderse los conceptos y métodos de la ingeniería del software. Solo entonces CASE proporcionará algún beneficio.

31.1 ¿QUÉ SIGNIFICA CASE?

El mejor taller de un artesano —sea mecánico, carpintero o ingeniero del software— tiene tres características fundamentales: (1) una colección de herramientas útiles que le ayudarán en todos los pasos de la construcción de un producto; (2) una disposición organizada que permitirá hallar rápidamente las herramientas y utilizarlas con eficacia; y (3) un artesano cualificado que entienda la forma de utilizar las herramientas de manera eficaz. Ahora es cuando los ingenieros del software reconocen que necesitan más herramientas y más variadas junto con un taller eficiente y organizado en el que puedan ubicarlas.

El taller de ingeniería del software se denomina un *entorno de apoyo integrado a proyectos* (que se describirá posteriormente en este capítulo), y el conjunto de herramientas que llena ese taller se denomina *ingeniería del software asistida por computadora* (CASE).

CASE proporciona al ingeniero la posibilidad de automatizar actividades manuales y de mejorar su visión general de la ingeniería. Al igual que las herramientas de ingeniería y de diseño asistidos por computadora que utilizan los ingenieros de otras disciplinas, las herramientas CASE ayudan a garantizar que la calidad se diseñe antes de llegar a construir el producto.

31.2 CONSTRUCCIÓN DE BLOQUES BÁSICOS PARA CASE



La ingeniería del software asistida por computadora puede ser tan sencilla como una única herramienta que preste su apoyo para una única actividad de ingeniería del software, o tan compleja como todo un entorno que abarque «herramientas», una base de datos, personas, hardware, una red, sistemas operativos, estándares, y otros mil componentes más. Los bloques de construcción de CASE se ilustran en la Figura 31.1. Cada bloque de construcción forma el fundamento del siguiente, estando las herramientas situadas en la parte superior del montón. Es interesante tener en cuenta que el fundamento de los entornos CASE efectivos tiene relativamente poco que ver con las herramientas de ingeniería del software en **sí.** Más bien, los entornos para la ingeniería del software se construyen con éxito sobre una arquitectura de entornos que abarca un hardware y un software de sistemas adecuados. Además, la arquitectura del entorno deberá tener en cuenta los patrones de trabajo humano que se aplicarán durante el proceso de ingeniería del software.

Cita:
Las herramientas CASE más valiosas son aquellas que contribuyen con información en el proceso de desarrollo.
Robert Dixon

Las arquitecturas del entorno, que constan de una plataforma hardware y de un soporte de sistema operativo (incluyendo software de red, gestión de la base de datos y servicios de gestión de objetos), establece los cimientos para un entorno CASE. Pero el entorno CASE en sí requiere otros bloques de construcción. Existe un conjunto de *servicios de portabilidad* que proporciona un puente entre las herramientas CASE, su marco de integración y la arquitectura del entorno. *El marco de integración* es un grupo de programas especializados que permiten a cada una de las herramientas comunicarse entre sí, para crear una base de datos del proyecto, y para mostrar el mismo aspecto al usuario final (el ingeniero del software). Los servicios de portabilidad permiten que las herramientas CASE y su marco de integración migren entre distintas plataformas del hardware y sistemas operativos sin un mantenimiento adaptativo significativo.

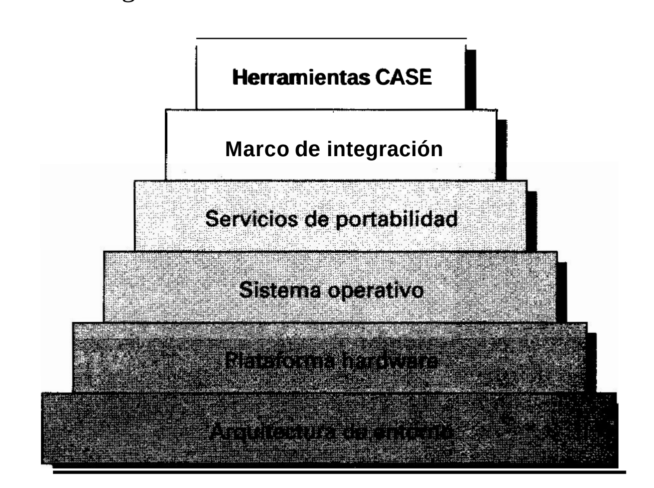


FIGURA 31.1. Bloques de construcción CASE.

Los bloques de construcción representados en la Figura 31.1 representan un fundamento completo para la integración de herramientas CASE. Sin embargo, la mayor parte de las herramientas CASE que se utilizan en la actualidad no han sido construidas empleando todos los bloques de construcción anteriormente descritos. De hecho, algunas herramientas siguen siendo las «soluciones puntuales». Esto es, una herramienta se utiliza para prestar apoyo en una actividad de ingeniería del software concreta (por ejemplo, modelado de análisis), pero esta herramienta no se comunica directamente con otras, no está unida a una base de datos del proyecto, y no forma parte de *un entorno integrado*

CASE (1-CASE). Aunque esta situación no es la ideal, se puede utilizar una herramienta CASE bastante eficiente, aunque se trate de una solicitud puntual.

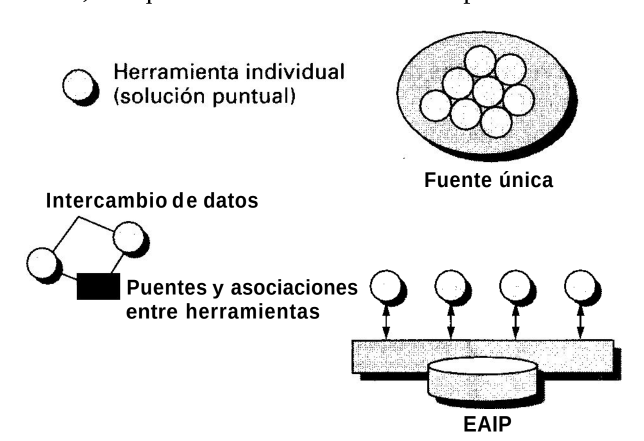


FIGURA 31.2. Opciones integradas.

Los niveles relativos de integración CASE se muestran en la Figura 31.2. En el extremo inferior del espectro de integración se encuentra la herramienta individual (solución puntual). Cuando las herramientas individuales proporcionan servicios para el intercambio de datos (como lo hacen la mayoría), el nivel de integración mejora ligeramente. Estas herramientas producen su salida en un formato estándar que deberá ser compatible con otras herramientas que sean capaces de leer ese formato. En algunos casos, los constructores de herramientas CASE complementarias trabajan juntos para formar un puente entre herramientas (por ejemplo, una herramienta de análisis y diseño que se enlaza con un generador de código). Mediante la utilización de este enfoque, la sinergia entre herramientas puede producir unos resultados finales que serían difíciles de crear empleando cada una de las herramientas por separado. La *integración de fuente única* se produce cuando un único vendedor de herramientas CASE integra una cierta cantidad de herramientas distintas y las vende en forma de paquete. Aunque este enfoque es bastante eficiente, la arquitectura cerrada de la mayoría de los entornos de fuente Única evita añadir fácilmente herramientas procedentes de otros fabricantes.



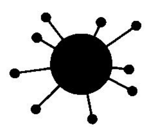
*las herramientas **CASE** de solución puntual pueden proporcionar un beneficio sustancia/, pero un equipa de software necesita herramientas que se comuniquen entre sí. las herramientas integradas ayudan a que el equipa desarrolle, organice y controle los productos del trabaja. Utilícelas.*

En el extremo superior del espectro de integración se encuentra *el entorno de apoyo integrado a proyectos integrado* (EAIP). Se han creado estándares en cada uno de los bloques de construcción descritos anteriormente. Los fabricantes de herramientas CASE utilizan los estándares EAIP para construir herramientas que sean compatibles con el EAIP, y que por tanto sean compatibles entre sí.

31.3 UNA TAXONOMÍA DE HERRAMIENTAS CASE

Existe un cierto número de riesgos que son inherentes siempre que se intenta efectuar una categorización de las herramientas CASE. Existe una sutil implicación que consiste en que para crear un entorno CASE efectivo se deberán implementar todas las categorías de herramientas —esto no es ni sencillo, ni cierto—. Cuando hay personas que creen que una herramienta pertenece a una categoría, se puede crear cierta confusión (o contradicción) al ubicar una herramienta específica dentro de otra categoría. Es posible que algunos lectores piensen que se ha omitido una categoría completa —eliminando por tanto un conjunto de herramientas completo e incluirlo así en el entorno CASE global—. Además, una categorización sencilla tiende a ser plana - esto es, no se muestra **la** interacción jerárquica de las herramientas o las relaciones que existen entre ellas—. A pesar de estos riesgos, es necesario crear una taxonomía de herramientas CASE —para comprender mejor la amplitud de CASE **y** para apreciar mejor en donde se pueden aplicar estas herramientas dentro del proceso del software—.

Las herramientas CASE se pueden clasificar por su función, por su papel como instrumentos para administradores o personal técnico, por su utilización en los distintos pasos del proceso de ingeniería del software, por la arquitectura del entorno (hardware y software) que les presta su apoyo, o incluso por su origen o coste [QED89]. La taxonomía que se presenta a continuación utiliza como criterio principal la función.



Herramientas **CASE**

Herramientas de ingeniería de procesos de negocio. Al modelar los requisitos de información estratégica de una organización, las herramientas de ingeniería de procesos de negocios proporcionan un «metamodelo» del cual se derivan sistemas de información específicos. En lugar de centrarse en los requisitos de una aplicación específica, estas herramientas CASE modelan la información de negocio cuando ésta se transfiere entre distintas entidades organizativas en el seno de una

compañía. El objetivo primordial de las herramientas de esta categoría consiste en representar objetos de datos de negocio, sus relaciones y la forma en que fluyen estos objetos de datos entre distintas zonas de negocio en el seno de la compañía.

Referencia cruzada
La ingeniería de procesos de negocios se trato en el Capítulo 10.

Modelado de procesos y herramientas de gestión. Si una organización trabaja por mejorar un proceso de negocio (o de software) lo primero que debe hacer es entenderlo. Las herramientas de modelado de procesos (llamadas también herramientas de *tecnología* ***de pro**-cesos*) se utilizan para representar los elementos clave del proceso de manera que sea posible entenderlo mejor. Estas herramientas también pueden proporcionar vínculos con descripciones de procesos que ayuden a quienes estén implicados en el proceso de comprender las tareas que se requieren para llevar a cabo ese proceso. Además, las herramientas de gestión de procesos pueden proporcionar vínculos con otras herramientas que proporcionan un apoyo para las actividades de proceso ya definidas.

Referencia cruzada
los elementos de procesos del software se tratan en el Capítulo 2.

Herramientas de planificación de proyectos. Las herramientas de esta categoría se centran en dos áreas primordiales: estimación de costes y de esfuerzos del proyecto de software y planificación de proyectos. Las herramientas de estimación calculan el esfuerzo estimado, la duración del proyecto y el número de personas recomendado para el proyecto. Las herramientas de planificación de proyectos hacen posible que el gestor defina todas las tareas del proyecto (la estructura de desglose de tareas), que cree una red de tareas (normalmente empleando una entrada gráfica), que represente las interdependencias entre tareas y que modele la cantidad de paralelismo que sea posible para ese proyecto.

Referencia cruzada
los técnicas de estimación se presentan en el Capitulo 5. los métodos de planificación se tratan en el Capítulo 7.

Herramientas de análisis de riesgos. La identificación de posibles riesgos y el desarrollo de un plan para mitigar, monitorizar y gestionar esos riesgos tiene una importancia fundamental en los proyectos grandes. Las herramientas de análisis de riesgos hacen posible que el gestor del proyecto construya una tabla de riesgos proporcionando una guía detallada en la identificación y análisis de riesgos.

Referencia cruzada
El análisis y gestión de riesgos se hato en el Capitulo 6.

Herramientas de gestión de proyectos. La planificación del proyecto y el plan del proyecto deberán ser rastreados y monitorizados de forma continua. Además, el gestor deberá utilizar las herramientas que recojan métricas que en Última instancia proporcionen una indicación de la calidad del producto del software. Las herramientas de esta categoría suelen ser extensiones de herramientas de planificación de proyectos.

Referencia cruzada
El seguimiento y lo monitorización se tratan en el Capitulo 7.

Herramientas de seguimiento de requisitos. Cuando se desarrollan grandes sistemas, las cosas «se derrumban». Es decir, el sistema proporcionado suele no satisfacer los requisitos especificados por el cliente. El objetivo de las herramientas de seguimiento es proporcionar un enfoque sistemático para el aislamiento de los requisitos, comenzando por el RFP del cliente o por la especificación. Las herramientas típicas de seguimiento de requisitos combinan una evaluación de textos por interacción humana, con un sistema de gestión de bases de datos que almacena y categoriza todos y cada uno de los requisitos del sistema que se «analizan» a partir de la RFP o especificación original.

Referencia cruzada
Los métodos de ingeniería de requisitos se tratan en el Capitulo 10.

Herramientas de métricas y de gestión. Las métricas del software mejoran la capacidad del gestor para controlar y coordinar el proceso de ingeniería del software y la capacidad del ingeniero para mejorar la calidad del software que se produce. Las métricas o herramientas de medidas actuales se centran en características de procesos y productos. Las herramientas orientadas a la gestión se sirven de métricas específicas del proyecto (por ejemplo, LDC/persona-mes, defectos por punto de función) que proporcionan una indicación global de productividad o de calidad. Las herramientas con orientación técnica determinan las métricas técnicas que proporcionan una mejor visión de la calidad del diseño o del código.

Referencia cruzada
Los métricas se presentan en los Capítulos 4, 19 y 24.

Herramientas de documentación. Las herramientas de producción de documentos y de autoedición pres-

tan su apoyo a casi todos los aspectos de la ingeniería del software, y representan una importante oportunidad de «aprovechamiento» para todos los que desarrollan software. La mayoría de las organizaciones dedicadas al desarrollo de software invierten una cantidad de tiempo considerable en el desarrollo de documentos, y en muchos casos el proceso de documentación en sí resulta bastante deficiente. Es frecuente que una organización de desarrollo de software invierta en la documentación hasta un 20 o un 30 por 100 de su esfuerzo global de desarrollo de software. Por esta razón, las herramientas de documentación suponen una oportunidad importante para mejorar la productividad.

Referencia cruzada
La documentación se estudia a lo largo de todo el libro. En SepaWeb se presentan más datos.

Herramientas de software de sistema. CASE es una tecnología de estaciones de trabajo. Por tanto, el entorno CASE deberá adaptarse a un software de sistema en red de alta calidad, al correo electrónico, a los tablones de anuncios electrónicos y a otras posibilidades de comunicarse.

Referencia cruzada
Consulte los Capítulos 27, 28 y 29 para un estudio limitado de estos temas.

Herramientas de control de calidad. La mayor parte de las herramientas CASE que afirman tener como principal interés el control de calidad son en realidad herramientas de métricas que hacen una auditoría del código fuente para determinar si se ajusta o no a ciertos estándares del lenguaje. Otras herramientas extraen métricas técnicas (Capítulos 19 y 24) en un esfuerzo por extrapolar la calidad del software que se está construyendo.

Referencia cruzada
GCS se presenta en el Capítulo 8.

Herramientas de gestión de bases de datos. El software de gestión de bases de datos sirve como fundamento para establecer una base de datos CASE (repositorio), que también se denominará base de datos del proyecto. Dada la importancia de los objetos de configuración, las herramientas de gestión de bases de datos para CASE pueden evolucionar a partir de los sistemas de gestión de bases de datos relacionales (SGBDR) para transformarse en sistemas de gestión de bases de datos orientadas a objetos (SGBDOO).

Referencia cruzada
El repositorio de software se estudia en el Capítulo 9.

Herramientas de gestión de configuración de software. La gestión de configuración de software (GCS) se encuentra en el núcleo de todos los entornos **CASE.** Las herramientas pueden ofrecer su asistencia en las cinco tareas principales de GCS —identificación, control de versiones, control de cambios, auditoría y contabilidad de estados—. La base de datos **CASE** proporciona el mecanismo de identificar todos los elementos de configuración y de relacionarlo con otros elementos; el proceso de control de cambios se puede implementar con ayuda de las herramientas especializadas; un acceso sencillo a los elementos de configuración individuales facilita el proceso de auditona; y las herramientas de comunicación CASE pueden mejorar enormemente la Contabilidad de estados (ofreciendo información acerca de los cambios a todos aquellos que necesiten conocerlos).

Referencia cruzada
Las actividades GCS en donde se incluyen la identificación, control de versiones, control de cambios, auditoría, y contabilidad de estados se estudian en el Capítulo 9.

Herramientas de análisis y diseño. Las herramientas de análisis y diseño hacen posible que el ingeniero del software cree modelos del sistema que vaya a construir. Los modelos contienen una representación de los datos, función y comportamiento (en el nivel de análisis), así como caracterizaciones del diseño de datos, de arquitectura, a nivel de componentes e interfaz[1]. Al efectuar una comprobación de consistencia y validez de los modelos, las herramientas de análisis y diseño proporcionan al ingeniero del software un cierto grado de visión en lo tocante a la representación del análisis, y le ayudan a eliminar errores antes de que se propaguen al diseño, o lo que es peor, a la propia implementación.

Referencia cruzada
El análisis y el diseño se estudian en las Partes Tercera y Cuarto de este libro.

Herramientas PRO/SIM. Las herramientas PRO/SIM (de construcción de prototipos y simulación) [NIC90] proporcionan al ingeniero del software la capacidad de predecir el comportamiento de un sistema en tiempo real antes de llegar a construirlo. Además, también le capacitan para desarrollar simulaciones del sistema de tiempo real, lo que permitirá que el cliente obtenga ideas acerca de su funcionamiento, comportamiento y respuesta antes de la verdadera implementación.

[1] La herramientas de diseño y de análisis orientado a objetos proporcionan representaciones análogas.

Referencia cruzada

La construcción de prototipos y la simulación se estudian brevemente en el Capitulo 10.

Herramientas de desarrollo y diseno de interfaz. Las herramientas de desarrollo y diseño de interfaz son en realidad un conjunto de herramientas de componentes de programas (clases) tales como menús, botones, estructuras de ventanas, iconos, mecanismos de desplazamiento de la pantalla, controladores de dispositivos, etc. Sin embargo, estos conjuntos de herramientas se están viendo sustituidos por herramientas de construcción de prototipos de interfaz que permiten una rápida creación en pantalla de interfaces de usuario sofisticadas, que se ajustan al estándar de interfaz que se haya adoptado para el software.

Referencia cruzada

los elementos del diseño de interfaz de usuario se presentan en el Capítulo 15.

Herramientas de construcción de prototipos. Se puede utilizar toda una gama de herramientas de construcción de prototipos. Los *generadores de pantallas* permiten al ingeniero del software definir rápidamente la disposición de la pantalla para aplicaciones interactivas. Otras herramientas de prototipos CASE más sofisticadas permiten la creación de un diseño de datos, acompañado por diseños e informes de la pantalla. Muchas herramientas de análisis y diseño son **más** extensas y proporcionan opciones de construcción de prototipos. Las herramientas PRO/SIM generan un diseño esquemático de código fuente en Ada y ***C*** para las aplicaciones de ingeniería (en tiempo real). Por último, una gama amplia de herramientas de cuarta generación poseen también características de construcción de prototipos.

Referencia cruzada

l o construcción de prototipos se estudio en los Capítulos 2 y 11.

Herramientas de programación. La categoría de herramientas de programación abarca los compiladores, editores y depuradores disponibles para apoyar a la mayoría de los lenguajes de programación convencionales. Además, en esta categoría también residen los entomos de programación orientados a objetos **(OO),** los lenguajes de cuarta generación, los entomos de programación gráfica, los generadores de aplicaciones y los lenguajes de consulta de bases de datos.

Herramientas de desarrollo de Webs. Las actividades asociadas a la ingeniería Web están apoyadas por una variedad de herramientas de desarrollo de WebApps. Entre estas herramientas se incluyen las que prestan ayuda en la generación de texto, gráficos, formularios, guiones, applets y otros elementos de una página Web.

Referencia cruzada

I—WEB se estudia en el Capítulo 29

Herramientas de integración y pruebas. En el directorio de herramientas de pruebas de software del Software Quality Engineering [SQE95], se definen las categorías de herramientas de pruebas siguientes:

- *adquisición de datos:* herramientas que adquieren los datos que se utilizarán durante la prueba;
- *medida estática:* herramientas que analizan el código fuente sin ejecutar casos de prueba;

Referencia cruzada

La comprobación del software se estudia en los Capítulos 17, 18 y 23 así como en los Capítulos 28 y 29.

- *medida dinámica:* herramientas que analizan el código fuente durante la ejecución;
- *simulación:* herramientas que simulan las funciones del hardware o de otros elementos externos;
- *gestión de pruebas:* herramientas que prestan su asistencia en la planificación, desarrollo y control de las pruebas;
- *herramientas de funcionalidad cruzada:* se trata de herramientas que atraviesan los límites de las categorías anteriores.

Debería tenerse en cuenta que muchas de las herramientas de prueba poseen características que abarcan dos categorías o más de las anteriormente mencionadas.

Herramientas de análisis estático. Las herramientas de análisis estático prestan su asistencia al ingeniero del software a efectos de derivar casos prácticos. En la industria se utilizan tres tipos distintos de herramientas estáticas de prueba: herramientas de prueba basadas en código; lenguajes de prueba especializados y herramientas de prueba basadas en requisitos. Las *herramientas de prueba basadas en código* admiten un código fuente (o LDP) como entrada, y llevan a cabo varios análisis de los que se obtiene la generación de casos de prueba. Los *lenguajes de prueba especializados* (por ejemplo, ATLAS) hacen posible que el ingeniero del software escriba especificaciones de prueba detalladas para describir todos los casos de prueba y la logística de su ejecución. Las *herramientas de prueba basadas en requisitos* aíslan los requisitos del usuario y sugieren los casos de prueba (o clases de prueba) que ejercitarán esos requisitos.

Referencia cruzada

Los métodos de comprobación de cap negra se estudian en el Capítulo 17.

Herramientas de análisis dinámico. Las herramientas de análisis dinámico interactúan con el programa que se esté ejecutando, prueban la cobertura de rutas, prueban las afirmaciones acerca del valor de variables específicas *e* instrumentan por otro lado el flujo de

ejecución del programa. Las herramientas dinámicas pueden ser intrusivas, o no intrusivas. Las *herramientas intrusivas* modifican el software que hay que comprobar mediante la inserción de sondas (instrucciones adicionales) y que efectúan las actividades mencionadas anteriormente. *Las herramientas de prueba no intrusivas* utilizan un procesador hardware por separado que funciona en paralelo con el procesador que contiene el programa que se está probando.

Herramientas de gestión de pruebas. Las herramientas de gestión de pruebas se utilizan para controlar y coordinar las pruebas del software por todos y cada uno de los pasos principales de las pruebas. Las herramientas de esta categoría gestionan y coordinan las pruebas de regresiones, efectúan comparaciones que determinan las diferencias entre la salida real y la esperada y realizan pruebas por lotes de programas con interfaces hombre-máquina interactivas. Además de las funciones indicadas anteriormente, muchas herramientas de gestión de pruebas sirven también como controladores de pruebas genéricos. Un controlador de pruebas lee uno o más casos de prueba de algún archivo de pruebas, aplica formato a los datos de prueba para que se ajusten a las necesidades del software que se está probando, e invoca entonces al software que es preciso probar.

Referencia cruzada

Las estrategias de prueba se estudian en el Capítulo 18.

Herramientas de pruebas cliente/servidor. El entorno C/S exige unas herramientas de pruebas especializadas que ejerciten la interfaz gráfica de usuario y los requisitos de comunicaciones en red para el cliente y el servidor.

Referencia cruzada

La prueba C/S se estudia en el Capítulo 28.

Herramientas de reingeniería. Las herramientas para el software heredado abarca un conjunto de actividades de mantenimiento que actualmente absorben un porcentaje significativo de todo el esfuerzo relacionado con el software. **La** categoría de herramientas de reingeniería se puede subdividir en las funciones siguientes:

- *herramientas de ingeniería inversa para producir especificaciones:* se toma el código fuente como entrada y se generan modelos gráficos de análisis y diseño estructurados, listas de utilización y más información sobre el diseño;

Referencia cruzada

Los métodos de reingeniería se estudian en el Capítulo 30.

- *herramientas de reestructuración y análisis de código:* se analiza la sintaxis del programa, se genera una gráfica de control de flujo y se genera automáticamente un programa estructurado; y
- *herramientas de reingeniería para sistemas on-line:* se utilizan para modificar sistemas de bases de datos on-line (por ejemplo, para convertir archivos **IDMS** o DB2 en un formato de entidades y relaciones).

Muchas de las herramientas anteriores se ven limitadas a lenguajes de programación específicos (aunque abarcan la mayoría de los lenguajes principales) y requieren un cierto grado de interacción con el ingeniero del software.

31.4 ENTORNOS CASE INTEGRADOS

Aunque se pueden obtener beneficios individualmente de las herramientas CASE que abarcan actividades de ingeniería del software por separado, la verdadera potencia de CASE solamente se puede lograr mediante la integración. Entre los beneficios del CASE integrado (1-CASE) se incluyen: (1) una transferencia regular de información (modelos, programas, documentos, datos) entre una herramienta y otra, y entre un paso de ingeniería y el siguiente; (2) una reducción del esfuerzo necesario para efectuar actividades globales tales como la gestión de configuración de software, el control de calidad y la producción de documentos; *(3)* un aumento del control del proyecto que se logra mediante una mejor planificación, monitorización y comunicación; (4) una mejor coordinación entre los miembros del personal que estén trabajando en grandes proyectos de software.



Ahora bien, 1-CASE también expone a desafíos significativos. En cada acción exige unas representaciones consecuentes de la información de la ingeniería del software; unas interfaces estandarizadas entre las herramientas, un mecanismo homogéneo para la comunicación entre el ingeniero del software y todas sus herramientas, y un enfoque efectivo que hará posible que 1-CASE se desplace a lo largo de distintas plataformas de hardware y distintos sistemas operativos. Los entornos 1-CASE generales han comenzado a surgir más lentamente de lo que inicialmente se esperaba. Sin embargo, los entornos integrados sí que existen, y con el paso de los años se van haciendo más poderosos.



la integración de los herramientas CASE exige uno base de datos que contenga las representaciones consecuentes de la información de la ingenierío del softwore.

El término integración implica tanto *combinación* como *cierre.* 1-CASE combina toda una gama de herramientas e información distintos de tal modo que hace posible el cierre de la comunicación entre las herramientas, entre personas y entre procesos de software. Las herramientas se integran de tal manera que la información de ingeniería del software esté disponible para todas las herramientas que se necesiten; la utilización se integra de tal modo que se proporciona un aspecto y una interacción común para todas las herramientas; y se integra una filosofía de desarrollo que aplica prácticas modernas y métodos ya probados.

Para definir la integración en el contexto del proceso del software, es necesario establecer un conjunto de requisitos para 1-CASE [FOR89a]. Un entorno CASE integrado debería:

- proporcionar un mecanismo para compartir la información de la ingeniería del software entre todas las herramientas dentro del entorno;
- hacer posible que un cambio de un elemento de información se siga hasta los demás elementos de información relacionados;
- proporcionar un control de versiones y una gestión de configuración general para toda la información de la ingeniería del software;
- permitir un acceso directo y no secuencia1 a cualquier herramienta del entorno;
- establecer un apoyo automatizado para el modelo de procesos de software que se haya seleccionado, integrando herramientas CASE y elementos de configuración del software en una estructura estándar de desglose de trabajo;
- permitir que los usuarios de cada una de las herramientas puedan experimentar con el aspecto e interacción de la interfaz hombre-máquina;

Referencia cruzada

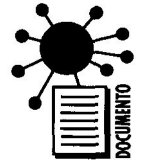
los **temas relacionados** con los procesos se **estudian** en los Capítulos 2,4 y 7. **Los ECs** se presentan en **el Capítulo 9.**

- dar soporte a la comunicación entre ingenieros del software; y
- recoger métricas tanto técnicas como de gestión que se puedan utilizar para mejorar el proceso y el producto.

Para alcanzar estos objetivos, cada uno de los bloques de construcción de una arquitectura (CASE —Fig. 31.1—) deberá encajar con los demás sin ningún tipo de limitación. Los bloques de construcción fundamentales —arquitectura del entorno, plataforma hardware y sistema operativo— deberán «unirse» a través de un conjunto de servicios de portabilidad a un marco de referencia de integración que alcance los objetivos indicados anteriormente.

31.5 LA ARQUITECTURA DE INTEGRACIÓN

Un equipo de ingeniería del software utiliza herramientas CASE, los métodos correspondientes y un marco de referencia de proceso con objeto de crear un conjunto de informaciones de ingeniería del software. El marco de referencia de integración facilita la transferencia de información desde y hacia ese conjunto de informaciones. Para lograr esto, deberán existir los componentes de arquitectura siguientes: la creación de una base de datos (para almacenar la información); la construcción de un sistema de gestión de objetos (para gestionar los cambios efectuados en la información); la construcción de un mecanismo de control de herramientas (para coordinar la utilización de herramientas CASE), y una interfaz de usuario que proporcione una ruta consecuente entre las acciones efectuadas por el usuario y las herramientas que están dentro del entorno. La mayoría de los modelos (por ejemplo, [FOR90], [SHA95] del marco de referencia de integración representan a estos componentes como si fueran capas. En la Figura **31.3.** se muestra un modelo sencillo del marco de referencia que representa únicamente los componentes indicados anteriormente.



Una lista de todos los elementos de información de la ingeniería del software puede encontrarse boio los ECs.

La *capa de interfaz del usuario* (Figura **3**1.1) incorpora un conjunto de herramientas de interfaz estandarizado, con un protocolo de presentación común. El kit de herramientas de interfaz contiene software para la gestión de la interfaz hombre-máquina, y una biblioteca de objetos de visualización. Ambos proporcionan un mecanismo consecuente para la comunicación entre la

interfaz y las herramientas CASE individuales. El *protocolo de presentación* es el conjunto de líneas generales que proporciona un mismo aspecto a todas las herramientas CASE. Las convenciones del diseño de pantalla, nombres y organización del menú, iconos, nombres de los objetos, utilización del teclado y del ratón, y el mecanismo para acceder a las herramientas se definen todos ellos como parte del protocolo de presentación.

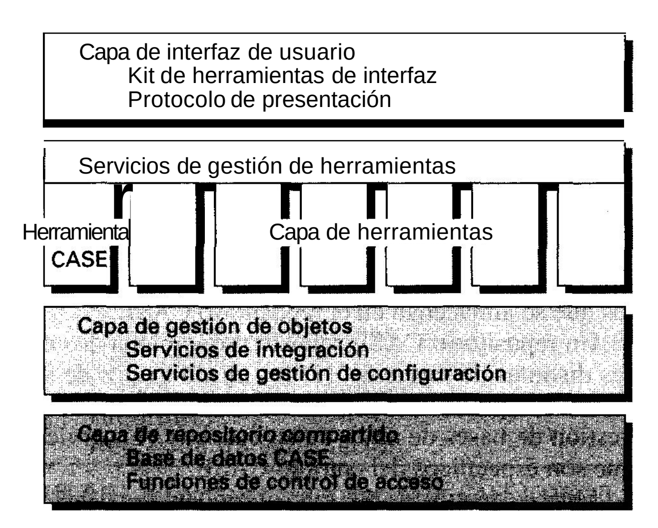


FIGURA 31.3. Modelo de arquitectura para el marco de referencia de integración.

La *capa de herramientas* incorpora un conjunto de servicios de gestión de herramientas con las herramientas CASE en sí. Los *servicios de gestión de herramientas* (SGH) controlan el comportamiento de las herramientas dentro del entorno. Si durante la ejecución de una o más herramientas se emplea la multitarea, **SGH** efectúa la sincronización y comunicación multitarea, coordina el flujo de información desde el repositorio y el sistema de gestión de objetos a las herramientas, realiza las funciones de seguridad y auditoría y recoge métricas acerca de la utilización de herramientas.



los recursos paro la integración de herramientas CASE y poro los Entornos de Integración de Ingeniería del software se pueden obtener en:
see.cs.flinders.edu.au/seweb/ti/

La *capa de gestión de objetos* (CGO) lleva a cabo las funciones de gestión de configuración que se describían en el Capítulo **8.** En esencia, el software de esta capa de la arquitectura de marco de referencia proporciona el mecanismo para la integración de herramientas. Cada herramienta CASE «se enchufa» en la capa de gestión de objetos. Al funcionar en conjunto con el repositorio CASE, la CGO proporciona los servicios de integración —un conjunto de módulos estándar que acoplan las herramientas con el repositorio—. Además, la OML proporciona los servicios de gestión de configuración haciendo posible la identificación de todos los objetos de configuración, llevando a cabo el control de versiones y proporcionando apoyo para el control de cambios, auditorías y contabilidad de estados.

La capa de repositorio compartido es la base de datos CASE y las funciones de control de acceso que hacen posible que la capa de gestión de objetos interactúe con la base de datos. La integración de datos se logra mediante las capas de gestión de objetos y de repositorio compartido que se estudian más adelante y con más profundidad en este mismo capítulo.

31.6 EL REPOSITORIO CASE

El *Diccionario Webster* define la palabra *repositorio* como «todo objeto o persona considerado centro de acumulación o almacenamiento». Durante las primeras fases de la historia del desarrollo del software, el repositorio era en realidad una persona -el programador que tenía que recordar la ubicación de toda la información relevante para un determinado proyecto de software, que tenía que recordar información que nunca se había escrito y que tenía que reconstruir la información que se había perdido—. Tristemente, la utilización de una persona como «centro de acumulación y almacenamiento» (aunque corresponda con la definición del diccionario) no funciona demasiado bien. En la actualidad, el repositorio es una «cosa» —una base de datos que actúa como centro tanto para la acumulación como para el almacenamiento de información de ingeniería del software—. El papel de la persona (del ingeniero del software) es interactuar con el repositorio empleando herramientas CASE integradas con él.

En este libro, se utiliza un determinado número de términos distintos para hacer alusión al lugar de almacenamiento de la información de la ingeniería del software: *base de datos* ***CASE,*** *base de datos del proyecto, base de datos de entorno de apoyo integrado a proyecto (EAIP), diccionario de requisitos* (una base de datos limitada) y repositorio, Aunque existen sutiles diferencias entre algunos de estos términos todos ellos hacen referencia al centro de acumulación y almacenamiento.

31.6.1. El papel del repositorio en 1-CASE

El repositorio de un entorno 1-CASE es el conjunto de mecanismos y de estructuras de datos que consiguen la

integración entre datos y herramientas, y entre datos y datos. Proporciona las funciones obvias de un sistema de gestión de bases de datos, pero, además, el repositorio lleva a cabo o precipita las funciones siguientes [FOR89b]:

- *integridad de datos:* incluye funciones para validar las entradas efectuadas en el repositorio, para asegurar la consistencia entre objetos relacionados, y para efectuar automáticamente modificaciones «en cascada» cuando un cambio efectuado en un objeto exige algún cambio en otros objetos relacionados con él;
- *información compartida:* proporciona un mecanismo para compartir información entre múltiples desarrolladores y entre múltiples herramientas; gestiona y controla el acceso multiusuario a los datos, y bloquea/desbloquea objetos para que los cambios no se superpongan inadvertidamente;

¿Cuales son las funciones que llevan a cabo los servicios que se acoplan con el repositorio CASE?

- *integración datos-herranzientas:* establece un modelo de datos al que pueden acceder todas las herramientas del entorno 1-CASE; controla el acceso a los datos, y lleva a cabo las funciones de gestión de configuración adecuadas;
- *integración duros-datos:* el sistema de gestión de bases de datos relaciona los objetos de datos de tal manera que se puedan alcanzar las demás funciones;
- *imposición de la metodología:* el modelo E-R de datos almacenado en el repositorio puede implicar un paradigma específico de ingeniería del software; como mínimo, las relaciones y los objetos definen un conjunto de pasos que se llevará a cabo para construir el contenido del repositorio;
- *estandarización de documentos:* la definición de objetos de la base de datos da lugar directamente a un enfoque estándar para la creación de documentos de ingeniería del software.

Para conseguir estas funciones, se define el repositorio en función de un metamodelo. El *metamodelo* determina la forma en que se almacena la información en el repositorio, la forma en que las herramientas pueden acceder a los datos y estos datos pueden ser visualizados por los ingenieros de software, el grado hasta el cual se puede mantener la seguridad e integridad de los datos, y la facilidad con que se puede ampliar el modelo ya existente para admitir nuevas necesidades [WEL89].

El metamodelo es la plantilla en la cual se sitúa la información de ingeniería del software. Una descripción detallada de estos modelos va más allá del alcance de este libro. Para más información, el lector interesado podría consultar [WEL89], [SHA95] y [GRI95].

31.6.2. Características y contenidos

Las características y contenido del repositorio se entienden especialmente bien examinándolo desde dos perspectivas: ¿Qué es lo que hay que almacenar en el repositorio, y qué servicios específicos son los que proporciona el repositorio? En general, los tipos de cosas que habrá que almacenar en el repositorio incluyen:

- el problema que hay que resolver;
- información acerca del dominio del problema;
- la solución del sistema a medida que va surgiendo;
- las reglas e instrucciones relativas al proceso de software (metodología) que se está siguiendo;
- el plan del proyecto, sus recursos y su historia;
- información acerca del contexto organizativo.

En la Tabla 31.1 se ha incluido una lista detallada de tipos de representaciones, documentos y otros productos que se almacenan en el repositorio CASE.

Un repositorio CASE robusto proporciona dos clases distintas de servicios: (1) los mismos tipos de servicios que puede uno esperar de cualquier sistema de gestiórfde bases de datos sofisticados; y (2) servicios que son específicos del entorno CASE.

Muchos requisitos del repositorio son iguales a los de las aplicaciones típicas que se construyen tomando como base un sistema de gestión de bases de datos de comercial (SGBD). De hecho, muchos de los repositorios CASE actuales hacen uso de un SGBD (normalmente relaciona1 u orientado a objetos) como la tecnología de gestión de datos básica. Entre las características de un SGBD que dan soporte a la gestión de información de desarrollo del software se incluyen las siguientes:

Almacenamiento de datos no redundante. Cada objeto es almacenado sólo una vez, aunque es accesible por todas las herramientas CASE siempre y cuando estas lo necesiten.

Acceso de alto nivel. Se implementa un mecanismo común de acceso a los datos de tal modo que no sea preciso duplicar las funciones de gestión de datos en todas las herramientas CASE.

¿Cuales son las características típicas de los SGBD que dan soporte a CASE ?

Independencia de datos. Las herramientas CASE y las aplicaciones destino se aíslan del almacenamiento físico para que no se vean afectadas cuando la configuración del hardware se cambie.

Control de transacciones. El repositorio implementa bloqueo de registros, admisiones de dos fases, registros de transacciones y procedimientos de recuperación para mantener la integridad de los datos cuando existen usuarios concurrentes.

Información de la empresa	Construcción
Estructura organizativa	Código fuente; código objeto
Análisis del área de negocios	Instrucciones de construcción del sistema
Funciones de negocios	Imágenes binarias
Reglas de negocios	Dependencias de configuración
Modelos de procesos (escenarios)	Información de cambios
Arquitectura de la información	**Validación y verificación**
Diseño de aplicaciones	Plan de comprobación; casos de prueba de los datos
Reglas de metodología	Guiones de comprobación de regresión
Representaciones gráficas	Resultados de comprobaciones
Diagramas de sistema	Análisis estadísticos
Estándares de denominación	Métricas de calidad de software
Reglas de integridad referenciales	**Información de gestión del proyecto**
Estructuras de datos	Planes de proyecto
Definiciones de proceso	Estructura de desglose de tareas
Definiciones de clase	Estimaciones; planificaciones
Arboles de menú	Carga de recursos; informe de problemas
Criterios de rendimiento	Solicitudes de cambios; informes de estado
Restricciones temporales	Información de auditoría
Definiciones de pantalla	**Documentación del sistema**
Definiciones de informes	Documentos de requisitos
Definiciones lógicas	Diseños internos/externos
Lógicas de comportamiento	Manuales de usuario
Algoritmos	
Reglas de transformación	

TABLA 31.1. Contenido de un repositorio CASE [FOR89B].

Seguridad. El repositorio proporciona mecanismos para controlar quién puede visualizar y modificar la información contenida en él.

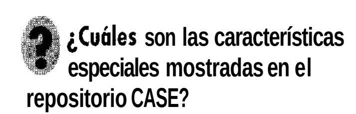
Consultas e *informes de datos ad hoc.* El repositorio permite acceder directamente a su contenido mediante una interfaz de usuario cómoda tal como SQL, o mediante un «navegador» *(browser)* orientado a formularios, haciendo posible un análisis definido por el usuario que va más allá de los informes estándar proporcionados por el conjunto de herramientas CASE.

Apertura. Los repositorios suelen proporcionar un mecanismo de importación/exportación sencillo que hace posible las cargas o transferencias de información al por mayor.

Soporte multiusuario. Un repositorio robusto deberá permitir que múltiples desarrolladores trabajen en una aplicación al mismo tiempo. Deberá gestionar el acceso concurrente a la base de datos mediante múltiples herramientas y por parte de múltiples usuarios, con arbitraje de accesos y con bloqueos en el nivel de archivos o registros. Para los entornos basados en redes, el soporte multiusuario implica también que el repositorio se podrá comunicar mediante interfaz con protocolos (agentes de solicitud de objetos) y servicios comunes de red.

El entorno CASE también efectúa demandas especiales con respecto al repositorio que van más allá de lo que está disponible directamente en un SGBD comercial. Entre las características especiales de los repositorios CASE se incluyen las siguientes:

Almacenamiento de estructuras de datos sofisticadas. El repositorio debe admitir tipos de datos complejos tales como diagramas, documentos y archivos, así como sencillos elementos de datos. Un repositorio también incluye un modelo de información (o metamodelo) que describe la estructura, relaciones y semántica de los datos almacenados en él. El metamodelo deberá poder ampliarse para dar cabida a representaciones nuevas y a una información organizativa únicas. El repositorio no solamente almacena modelos y descripciones de los sistemas en desarrollo, sino que los metamodelos asociados (esto es, una información adicional que describe la información de ingeniería del software en **sí,** tal como el momento en que se ha creado un componente de diseño concreto, su estado actual y la lista de componentes de los cuales depende).



Imposición de una integridad. El modelo de información del repositorio contiene también reglas, o políticas, que describen reglas de negocios válidas y otras restricciones y requisitos acerca de la información que se inserta en el repositorio (directamente o a través de una herramienta CASE). Es posible emplear un servicio llamado *disparador* para activar las reglas asociadas a un objeto siempre que este sea modificado, lo cual hace posible verificar la validez de los modelos de diseño en tiempo real.



Un manual de aprendizaje detallado y una lista de recursos para repositorios OO (que se pueden utilizar para entornos CASE) se pueden encontrar en la dirección **mini.net/cetus/oo_db_systems_1.html**

Interfaz de herramientas ricas en términos semánticos. El modelo de información del repositorio (el metamodelo) contiene una semántica que hace posible que toda una gama de herramientas interpreten el significado de los datos almacenados en el repositorio. Por ejemplo, un diagrama de flujo de datos creado mediante una herramienta CASE se almacena en el repositorio en un formulario basado en el modelo de información e independiente de toda representación interna que pueda utilizar la herramientas en **sí.** Entonces otra herramienta CASE puede interpretar el contenido del repositorio y utilizar la información cuando la necesite para **su** tarea. De este modo, la semántica almacenada en un repositorio permite compartir datos entre una gran variedad de herramientas, a diferencia de las conversiones específicas entre herramientas, o entre «puentes».

Gestión de procesosiproyectos. Un repositorio contiene información no sólo acerca de la aplicación de software en **sí,** sino también acerca de las características de cada proyecto en particular, y del proceso general de la organización para el desarrollo del software (fases, tareas y productos). Esto abre posibilidades para la coordinación automatizada de la actividad de desarrollo técnico con la actividad de gestión del proyecto. Por ejemplo, la actualización del estado de las tareas de proyectos se podría efectuar de forma automática o bien como un producto derivado de la utilización de herramientas CASE. La actualización de estado resultará muy fácil para los desarrolladores, sin tener que abandonar el entorno de desarrollo normal. La asignación de tareas y consultas también se puede gestionar por correo electrónico. Los informes de problemas, las tareas de mantenimiento, las autorizaciones de cambios, y los estados de reparación se pueden coordinar y monitorizar mediante herramientas que acceden al repositorio.

Las siguientes características del repositorio son abarcadas todas ellas por la gestión de configuración del software (Capítulo 9). Se vuelven a examinar aquí para hacer hincapié en su interrelación con los entornos 1-CASE.

Versiones. A medida que avanza un proyecto, se irán creando muchas versiones de productos individuales. El repositorio deberá ser capaz de guardar todas estas versiones para hacer posible una gestión efectiva de las versiones de los productos y para permitir que los desarrolladores vuelvan a las versiones anteriores durante la comprobación y depuración.



El repositorio CASE deberá ser capaz de controlar una amplia variedad de tipos de objetos entre los que se incluyen texto, gráficos, mapas de bits, documentos complejos y objetos Únicos como definiciones de pantalla y de informes, archivos de objetos, datos de comprobación y resultados. Un repositorio maduro rastrea las versiones de objetos con niveles arbitrarios de granularidad, por ejemplo, se puede rastrear cada definición de datos o agrupamiento de módulos.

Para dar apoyo al desarrollo paralelo, el mecanismo de control de versiones deberá permitir múltiples derivados (variantes) a partir de un solo predecesor. Así pues, un desanollador podrá estar trabajando al mismo tiempo, en dos soluciones posibles para un problema de diseño generadas las dos desde el punto de partida.

Seguimiento de dependencias y gestión de cambios. El repositorio gestiona una amplia variedad de relaciones entre los elementos de datos almacenados en él. Entre estas se cuentan las relaciones entre entidades y procesos de la empresa, entre las partes de un diseño de aplicación, entre componentes del diseño y la arquitectura de la información de la empresa, entre elementos de diseño y productos, etc. Algunas de las relaciones son meramente asociaciones, y algunas son dependencias o relaciones obligatorias. El mantenimiento de estas relaciones entre objetos de desarrollo se denomina *administración de enlaces.*



La capacidad del repositorio para hacer seguimiento de las relaciones entre objetos de la configuración es una de las características más importantes. El impacto del cambio se puede rastrear si se dispone de esta característica.

La capacidad de seguir la pista de todas estas relaciones es crucial para la integridad de la información almacenada en el repositorio y para la generación de productos basados en ella, y es una de las contribuciones más importantes que efectúa el concepto de repositorio para la mejora del proceso de desarrollo del software. Entre las muchas funciones que apoya la gestión de enlaces se cuenta la capacidad de identificar y estimar los efectos del cambio. **A** medida que los diseños evolucionan para satisfacer nuevos requisitos, la capacidad de identificar todos los objetos que podrían verse afectados hace posible una estimación más precisa del coste, del tiempo no operativo, y del grado de dificultad. También se ayuda a evitar efectos colaterales que en caso contrario darían lugar a defectos y a fallos del sistema.

La gestión de enlaces ayuda al mecanismo de repositorio para asegurar que la información del diseño sea correcta manteniendo sincronizadas las partes de un diseño.

Por ejemplo, si se modifica un diagrama de flujo de datos, el repositorio puede detectar **si** los diccionarios

de datos relacionados, definiciones de pantallas y módulos de códigos también requieren modificación y puedan llamar la atención del desarrollador los componentes afectados.

Seguimiento de requisitos. Esta función especial depende de la gestión de enlaces y proporciona la capacidad de hacer seguimiento de los componentes de diseño y de los productos derivados que proceden de una especificación de requisitos específica (seguimiento progresivo). Además, proporciona la capacidad de identificar cuáles son los requisitos que generaron cualquier producto derivado (seguimiento regresivo).

Gestión de configuración. Una función de gestión de configuración que trabaja muy cerca de las funciones de versiones y gestión de enlaces para hacer seguimiento de una serie de configuraciones que representarán hitos específicos del proyecto o de versiones de producción. La gestión de versiones proporciona las versiones necesarias, y la gestión de enlaces hace seguimiento de las interdependencias.

Seguimiento de auditoría. El seguimiento de auditoría establece información extra acerca de cuándo, por qué, y por quién son efectuados los cambios. La información acerca de las fuentes de las modificaciones se pueden introducir en forma de atributos de objetos específicos del repositorio. Un mecanismo disparador del repositorio resultará útil para solicitar al desarrollador o a la herramienta que esté utilizando que inicie la introducción de información de auditoría.(tal como la razón del cambio) siempre que se modifique un elemento de diseño.

RESUMEN

Las herramientas de ingeniería del software asistida por computadora abarcan todas las actividades del proceso del software y también aquellas actividades generales que se aplican a lo largo de todo el proceso. CASE combina un conjunto de bloques de construcción que comienzan en el nivel del hardware y del software de sistema operativo y finalizan en las herramientas individuales.

En este capítulo se ha tenido en consideración una taxonomía de herramientas CASE. Las categorías abarcan tanto las actividades de gestión como las técnicas, e incluyen la mayor parte de las áreas de aplicación del software. Todas las categorías de herramientas se han considerado «soluciones puntuales».

El entorno 1-CASE combina mecanismos de integración para datos, herramientas e interacción hombre-computadora. La integración de datos se puede conseguir mediante el intercambio directo de información, mediante estructuras de archivos comunes, mediante datos compartidos o interoperabilidad, o a través de la utilización de un repositorio 1-CASE completo. La integración de herramientas se puede diseñar de forma personalizada por parte de fabricantes que trabajan a la vez, o bien se puede lograr mediante un software de gestión que se proporcione como parte del repositorio.

La integración entre hombre y computadora se logra mediante estándares de interfaz que se están volviendo cada vez más comunes a lo largo y ancho de toda la industria. Para facilitar la integración de los usuarios con las herramientas, de las herramientas entre sí, de las herramientas con los datos y de los datos con otros datos se diseña una arquitectura de integración.

Se ha aludido al repositorio CASE con el nombre de «bus de software». La información pasa por él, y va circulando de herramienta en herramienta a medida que progresa el proceso de software. Pero el repositorio es mucho más que un «bus». También se trata de un lugar de almacenamiento que combina sofisticados mecanismos para integrar herramientas CASE mejorando consiguientementeel proceso mediante el cual se desarrolla el software. El repositorio es una base de datos relacional u orientada a objetos que es «el centro de acumulación y almacenamiento» de la información de ingeniería del software.

REFERENCIAS

[FOR89a] Forte, G., «In Search of the Integrated Environment», *CASE Outlook,* Marzo/Abril de 1989, pp. 5-12.

[FOR89b] Forte, G., «Rally Round the Repository», *CASE Outlook,* Diciembre de 1989, pp. 5-27.

[FOR90] Forte, G., «Integrated CASE: A Definition», *Proc. 3rd Annual TEAMWORKERS Intl. User's Group Conference*, Cadre Technologies, Providence, IR, Marzo de 1990.

[GRI95] Griffen, J., «Repositories: Data Dictionary Descendant Can Extend Legacy Code Investment», *Application Development Trends,* Abril de 1995, pp. 65-71.

[QED89] *CASE: The Potential and the Pitfalls,* QED Information Sciences, Inc., Wellsley, **MA,** 1989,

[SHA95] Sharon, D., y R. Bell, «Tools that Bind: Creating Integrated Environments», *IEEE Software*, Marzo de 1995, pp. 76-85.

[SQE95] *Testing Tools Reference Guide,* Software Quality Engineering, Jacksonville, FL, 1995.

[WEL89] Welke, R.J. «Meta Systems on Meta Models», *CASE Outlook,* Diciembre de 1989, pp. 35-45.

PROBLEMAS Y PUNTOS A CONSIDERAR

31.1. Haga una lista de todas las herramientas de desarrollo de software que utilice. Organícelas de acuerdo con la taxonomía presentada en este capítulo.

31.2. Empleando las ideas introducidas en los Capítulos 14 y 16, ¿cómo cree usted que deberían de construirse los servicios de portabilidad?

31.3. Construye un prototipo en papel para una herramienta de gestión de proyectos que abarque las categorías indicadas en la Sección 31.3. Utilice la Segunda Parte de este libro para obtener información adicional.

31.4. Lleve a cabo una investigación acerca de los sistema de gestión de bases de datos orientados a objetos. Estudie por qué un SGBDOO sería ideal para herramientas GCS.

31.5. Recoja la información de productos de al menos tres herramientas CASE en una categona especificada por su profesor. Desarrolle una matriz que compare las características.

31.6. ¿Existen situaciones en las cuales las herramientas de comprobación dinámicas sean «la Única forma posible»? De ser así, ¿cuáles son?

31.7. Describa otras actividades humanas en las cuales la integración de un conjunto de herramientas haya proporcionado un mayor beneficio que la utilización de cada una de esas herramientas de forma individual. No utilice ejemplos de computación.

31.8. Describa lo que quiere decir la integración entre herramientas y datos con sus propias palabras.

31.9. En diferentes lugares de este capítulo se utilizan los términos metamodelos y metadatos. Describa lo que significan estos términos empleando sus propias palabras.

31.10. ¿Se le ocurre algún elemento de configuración adicional que pudiera incluirse entre los elementos del repositorio mostrados en la Tabla 31.1? Haga una lista.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

En los años ochenta y principios de los noventa se publicaron varios libros sobre CASE en un esfuerzo de sacar provecho del alto grado de interés de la industria en ese momento. Entre las primeras ofertas que todavía disfrutan de valor se encuentran:

Bergin, T. et al., *Computer-Aided Software Engineering: Issues and Trendsfor the 1990s and Beyond,* Idea Group Publishing, 1993.

Braithwaite, K.S., *Application Development Using CASE Tools,* Academic Press, 1990.

Brown, A. W., D.J. Carney y E.J. Morris, *Principles oj CASE Tool Integration,* Oxford University Press, 1994.

Clegg, D., y R. Barker, *CASE Method Fast-Track: A RAD Approach,* Addison Wesley, 1994.

Lewis, T.G., *Computer-Aided Software Engineering,* Van Nostrand-Reinhold, 1990.

Mylls, *R., Information Engineering; CASE Practices and Techniques,* Wiley, 1993.

En la antología de Chifofsky *(Computer-Aided Software Engineering,* IEEE Computer Society, 2.ª Ed., 1992) se encuentra una colección útil de los primeros trabajos sobre CASE y entornos de desarrollo de software. Muller y sus colaboradores *(ComputerAided Software Engineering,* Kluwer Academic Publishers, 1996) han editado una colección de descripciones de investigación CASE a mediados de los noventa. La mejores fuentes de información actual sobre herramientas CASE se encuentran en Intemet, en publicaciones técnicas periódicas y en boletines informativos de industria.

El estándar 1209 IEEE (*Evaluation and Selection of CASE Tools)* presenta un conjunto de líneas generales para evaluar las herramientas CASE para los «procesos de gestión de proyectos, procesos de predesarrollo, procesos de desarrollo, procesos de postdesarrollo y procesos integrales». Un informe detallado de Wallnau y Feiler *(Tool Integration and Environment Architectures, Software Engineering Institutc,* CMU/SEI-9 1-TR-11, Mayo de 1991), aunque esté fechado, sigue siendo uno de los mejores tratamientos sobre entornos CASE fácilmente disponibles.

Una amplia variedad de fuentes de información sobre CASE está disponible en Intemet. Una amplia lista actualizada de referencias a sitios (páginas) web se puede obtener en **http:/www.pressman5.com.**

CAPÍTULO

32 PERSPECTIVAS FUTURAS

EN los 31 capítulos anteriores, se ha ido explorando un proceso de ingeniería del software. Se han presentado procedimientos de gestión y métodos técnicos, principios básicos y técnicas especializadas, actividades orientadas a personas y tareas adecuadas para su automatización, notación de papel y lápiz y herramientas CASE. Se ha afirmado que la medida, la disciplina y un especial interés por la calidad darán lugar a un software que va a satisfacer las necesidades del usuario, a un software fiable, a un software mantenible y a un software *mejor*. Y, sin embargo, nunca se ha prometido que la ingeniería del software fuera la panacea.

A medida que vamos avanzando hacia el comienzo de un nuevo siglo, la tecnología de software y de sistemas siguen siendo un desafío para todo profesional del software y para toda compañía que construya sistemas basados en computadoras. Aunque esto se escribió hace más de una década, Max Hopper [HOP90] describe el estado actual de los acontecimientos:

> Dado que **los** cambios de la tecnología de la información cada vez se producen más rápidamente y son implacables, y dado que las consecuencias de quedarse atrás **son** irreversibles, las compañías tienen que dominar esta tecnología o morir. Hay que pensar que se trata del molino de la tecnología. Las compañías tendrán que correr cada vez más deprisa para estar a la altura.

Los cambios de la tecnología de la ingeniería del software son, en efecto, «rápidos e implacables», pero al mismo tiempo el progreso suele ser bastante lento. Pero una vez que se toma la decisión de adaptar un método nuevo (o una herramienta nueva), la decisión de llevar a cabo la formación necesaria para comprender su aplicación, y la decisión de introducir la tecnología en la cultura de desarrollo del software, ya ha surgido algo nuevo (e incluso mejor) y comienza de nuevo el proceso.

VISTAZO RÁPIDO

¿Qué es? El futuro nunca es fácil de predecir: los entendidos y expertos de la industria no se resisten. El futuro estaba plagado de carcasas de tecnologías nuevas y excitantes que realmente nunca llegaron a ser eso (apesar del bombo que tuvieron), y que adquirieron forma con tecnologías más modernas que de alguna manera modificaron su dirección y extensión. Por lo tanto no pretendemos predecir el futuro sino que estudiaremos algunos temas que hay que tener en cuenta para entender los cambios futuros de la ingeniería del software y del mismo software.

¿Quién lo hace? Todos.

¿Por qué es importante? ¿Por qué los reyes antiguos contrataban adivinos?, ¿por qué las grandes empresas multinacionales contratan a otras empresas asesoras y a grandes pensadores paru preparar los pronósticos?, ¿por qué un gran porcentaje de lectores están pendientes de los horóscopos? Queremos saber lo que va a ocurrir para estar preparados.

¿Cuáles son los pasos? No hay una fórmula para predecir el futuro. Lo hemos intentado recogiendo datos y organizándolos con el fin de proporcionar una información útil; examinando las asociaciones sutiles para extraer el conocimiento y, a partir de éste, sugerir posibles apariciones que predecirán cómo será todo en el futuro.

¿Cuál es el producto obtenido? Una visión del término futuro que pueda o no ser correcta.

¿Cómo puedo estar seguro de que lo he hecho correctamente? Predecir el futuro es un arte, no una ciencia. De hecho, es bastante extraño cuando se ha hecho una predicción seria, que ésta sea absolutamente verdadera o inequívocamente errónea (con la excepción, afortunadamente, de las predicciones sobre el fin del mundo). Se buscan tendencias y se intenta extrapolarlas al futuro. Solamente se puede evaluar la corrección de esta extrapolación a medida que pasa el tiempo.

En este capítulo se examinan las tendencias futuras. Nuestro intento no es explorar todas las áreas de investigación que resulten prometedoras. Tampoco lo es mirar en una «bola de cristal» y pronosticar el futuro, más bien se intentará explorar el ámbito del cambio y la forma en que el cambio en sí va a afectar al proceso del software en los años venideros.

32.1 IMPORTANCIA DEL SOFTWARE —SEGUNDA PARTE—

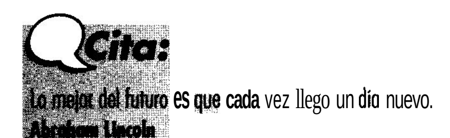
La importancia del software de computadora se puede enunciar de muchas formas. En el Capítulo 1, el software se caracterizaba como *diferenciador*. La función proporcionada por el software es lo que diferencia a los productos, sistemas y servicios, y proporciona una ventaja competitiva en el mercado. Sin embargo, el software es más que un diferenciador. Los programas, documentos y datos que constituyen el software ayudan a generar la mercancía más importante que pueda adquirir cualquier individuo, negocio o gobierno —la información—. Pressman y Herron [PRE91] describen el software de la forma siguiente:

> El software de computadora es una de las pocas tecnologías clave que tendrán un impacto significativo en los años 90. Se trata de un mecanismo para automatizar negocios, industrias y gobiernos, un medio para transferir nuevas tecnologías, un método para adquirir experiencia valiosa que otras personas puedan utilizar, un medio para diferenciar los productos de una compañía de los productos de los de sus competidores, y una ventana que permite examinar el conocimiento colectivo de una corporación. El software es crucial para casi todos los aspectos del negocio. Pero de muchas maneras, el software es también una tecnología oculta. Encontramos el software (frecuentemente, sin darnos cuenta) cuando nos desplazamos hasta nuestro trabajo, cuando efectuamos cualquier compra, cuando nos detenemos en el banco, cuando hacemos una llamada telefónica, cuando visitamos al médico o cuando realizamos cualquiera de los cientos de actividades diarias que reflejan la vida moderna.
>
> El software está en todas partes y, sin embargo, hay muchas personas en puestos de responsabilidad que tienen poca o ninguna comprensión de lo que realmente es, como se construye, o de lo que significa para las instituciones que lo controlan (y a las que controla). Y, lo que es más importante, tienen muy poca idea de los peligros y oportunidades que este software ofrece.

La omnipresencia del software nos lleva a una conclusión sencilla: siempre que una tecnología tiene un impacto amplio —un impacto que puede salvar vidas o ponerlas en peligro, construir negocios o destruirlos, informar a los jefes de gobierno o confundirlos—, es preciso «manejarla con cuidado».

32.2 EL ÁMBITO DEL CAMBIO

Los cambios en la informática durante los últimos 50 años han sido controlados por los avances en las «ciencias experimentales duras» —física, química, ciencia de materiales e ingeniería—. Durante las próximas décadas, los avances revolucionarios en la informática serán dirigidos por las ciencias no experimentales suaves» —psicología humana, biología, neurofisiología, sociología, filosofía y otras—. El período de gestación de las tecnologías informáticas que se puede derivar de estas disciplinas es muy difícil de predecir.



Es posible que la influencia de las ciencias no experimentales ayude a moldear la dirección de la investigación informática en las ciencias experimentales. Por ejemplo, el diseño del las «computadoras futuras» puede que sea dirigido más por un entendimiento de la psicología del cerebro que por el entendimiento de la microelectrónica convencional.

Los cambios que afectarán a la ingeniería del software durante la próxima década se verán influenciados por cuatro fuentes simultáneas: (1) las personas que realizan el trabajo; (2) el proceso que aplican; **(3)** la naturaleza de la información, y (4) la tecnología informática subyacente. En las secciones siguientes, cada uno de estos componentes —personas, proceso, información y tecnología— se estudiarán detalladamente.

32.3 LAS PERSONAS Y LA FORMA EN QUE CONSTRUYEN SISTEMAS

El software necesario para los sistemas de alta tecnología cada vez son más difíciles a medida que van pasando los años y el tamaño de los programas resultantes incrementa de manera proporcional. El crecimiento rápido del tamaño del programa «promedio» nos presentará unos cuantos problemas si no fuera por el simple hecho de que: a medida que el programa aumenta, el número de personas que deben trabajar en él también aumenta.

La experiencia indica que a medida que aumenta el número de personas de un equipo de proyecto de software, es posible que la productividad global del grupo sufra. Un método para resolver este problema es crear un número de equipos de ingeniería del software, por el que se compartimentalicen las personas en grupos de trabajo individuales. Sin embargo, a medida que el número de equipos de ingeniería del software aumenta, la comunicación entre ellos es tan difícil y larga como la comunicación entre individuos. Es aún peor, la comunicación (entre individuos o equipos) —es decir, es demasiado tiempo el que se pasa transfiriendo poco con-

tenido de información, y toda esta información importante suele «perderse entre las grietas»—.

Si la comunidad de ingeniería del software va a tratar el dilema de la comunicación de manera eficaz, el futuro de los ingenieros deberá experimentar cambios radicales en la forma en que los individuos y los equipos se comunican entre **sí.** El correo electrónico, los tablones de anuncios y los sistemas de videoconferencia son los lugares comunes como mecanismos de conexión entre muchas personas de una red de información. La importancia de estas herramientas en el contexto del trabajo de la ingeniería del software no se puede sobrevalorar. Con un sistema de correo electrónico o de tablón de anuncios eficaz, el problema que se encuentra un ingeniero del software en la ciudad de Nueva York, se puede resolver con la ayuda de un compañero en Tokio. En realidad, los tablones de anuncios y los grupos de noticias especializados se convierten en los depósitos de conocimiento que permiten recoger un conocimiento colectivo de un grupo grande de técnicos para resolver un problema técnico o un asunto de gestión.

Cita:

El choque futuro es el tremendo agobio y la desorientación que inducimos en los individuos sometiéndolos a demasiados cambios en períodos muy cortos de tiempo.

Alvin Toffler

El vídeo personaliza la comunicación. Y lo mejor es que hace posible que compañeros en lugares diferentes (o en continentes diferentes) se «encuentren» regularmente. Pero el vídeo también proporciona otra ventaja: se puede utilizar como depósito para conocimiento sobre el software y para los recién llegados a un proyecto.

La evolución de los agentes inteligentes también cambiará los patrones de trabajo de un ingeniero del software ampliando dramáticamente las herramientas del software. Los agentes inteligentes mejorarán la habilidad de los ingenieros mediante la comprobación de los productos del trabajo utilizando el conocimiento específico del dominio, realizando tareas administrativas, llevando a cabo una investigación dirigida y coordinando la comunicación entre personas.

Finalmente, la adquisición de conocimiento está cambiando profundamente. En Intemet, un ingeniero del software puede subscribirse a un grupo de noticias centrado en áreas de tecnología de inmediata preocupación. Una pregunta enviada a un grupo de noticias provoca respuestas de otras partes interesadas desde cualquier lugar del globo. La World Wide Web proporciona al ingeniero del software la biblioteca más grande del mundo de trabajos e informes de investigación, manuales, comentarios y referencias de la ingeniería del software[1].

Si la historia pasada se puede considerar un indicio, es justo decir que las mismas personas no van a cambiar. Sin embargo, las formas en que se comunican, el entorno en el que trabajan, la forma en que adquieren el conocimiento, los métodos y herramientas que utilizan, la disciplina que aplican, y por tanto, la cultura general del desarrollo del software **sí** cambiarán significativa y profundamente.

32.4 EL «NUEVO» PROCESO DE INGENIERÍA DEL SOFTWARE

Es razonable caracterizar las dos primeras décadas de la práctica de ingeniería del software como la era del «pensamiento lineal». Impulsado por el modelo de ciclo vital clásico, la ingeniería del software se ha enfocado como una actividad en la cual se podían aplicar una serie de pasos secuenciales en un esfuerzo por resolver problemas complejos. Sin embargo, los enfoques lineales del desarrollo del software van en contra de la forma en que se construyen realmente la mayoría de los sistemas. En realidad, los sistemas complejos evolucionan de forma iterativa, e incluso incremental. Por esta razón, un gran segmento de la comunidad de la ingeniería del software se está desplazando hacia modelos evolutivos para el desarrollo del software.

Los modelos de proceso evolutivos reconocen que la incertidumbre domina la mayoría de los proyectos; que las líneas temporales suelen ser imposibles y cortas; y que la iteración proporciona la habilidad de dar una solución parcial, aunque un producto completo no es posible dentro del marco de tiempo asignado. **Los** modelos evolutivos hacen hincapié en la necesidad de productos de trabajo incrementales, análisis de riesgos, planificación y revisión de planes, y realimentación que provenga del cliente.

Cita:

La mejor preparación de buen trabajo para mañana es hacer un buen trabajo hoy.

Elbert Hubbard

¿Qué actividades deberán de poblar el proceso evolutivo? A lo largo de la década pasada, el *Modelo de madurez de capacidad* desarrollado por el Software Engineering Institute **(SEI)** [PAU93] ha tenido un impacto apreciable sobre los esfuerzos por mejorar las prácticas de ingeniería del software. El MMC ha dado lugar a muchos debates (por ejemplo, [BOL91] y [GIL96]), y sin embargo proporciona una buena indicación de los

[1] El sitio Web SEPA puede proporcionar enlaces electrónicos con las materias más importantes que se presentan en este libro.

atributos que deben existir cuando se pone en práctica una buena ingeniería del software.

Las tecnologías de objetos, junto con la ingeniería del software (Capítulo 27) son un brote de la tendencia hacia los modelos de proceso evolutivos. Ambos tendrán un impacto profundo sobre la productividad de desarrollo del software y sobre la calidad del producto. La reutilización de componentes proporciona beneficios inmediatos y convincentes. Cuando la reutilización se une a las herramientas CASE para los prototipos de una aplicación, los incrementos del programa se pueden construir mucho más rápidamente que mediante la utilización de enfoques convencionales. La construcción de prototipos arrastran al cliente al proceso. Por tanto es probable que clientes y usuarios se impliquen más en el desarrollo del software. Esto, a su vez, puede llevar a una satisfacción mayor del usuario final y a una calidad mejor del software global.

El crecimiento rápido de las aplicaciones basadas en Web (WebApps) está cambiando tanto en el proceso de la ingeniería del software como en sus participantes. De nuevo, nos encontramos con un paradigma incremental y evolutivo. Pero en el caso de las WebApps, la inmediatez, seguridad y estética se están convirtiendo en las preocupaciones dominantes. Un equipo de ingeniería de Web mezcla técnicos con especialistas de contenido (por ejemplo, artistas, músicos, videógrafos) para construir una fuente de información para una comunidad de usuarios grande e impredecible. El software que ha surgido del trabajo de la ingeniería de Web ya ha dado como resultado un cambio radical tanto económico como cultural. Aunque los conceptos y principios básicos tratados en este libro son invariables, el proceso de ingeniería del software deberá adaptarse para llegar a acoplarse a la Web.

32.5 NUEVOS MODOS DE REPRESENTAR LA INFORMACIÓN

A lo largo de las dos últimas décadas, se ha producido una sutil transición en la terminología que se utiliza para describir el trabajo de desarrollo de software realizado para la comunidad de negocios. Hace treinta años, el término *procesamiento de datos* era la frase operativa para describir la utilización de computadoras en un contexto de negocio. En la actualidad, el proceso de datos ha dado lugar a otra frase —tecnología *de la información*— que significa lo mismo pero presenta un sutil cambio de nuestro centro de atención. La importancia ya no está meramente en procesar grandes cantidades de datos, sino más bien en extraer una información significativa a partir de estos datos. Evidentemente, éste ha sido siempre el objetivo, pero el cambio de la terminología refleja un cambio bastante más importante en **la** filosofía de la gestión.

Cuando en la actualidad se describen las aplicaciones de software, las palabras *datos* e *información* aparecen en numerosas ocasiones. La palabra *conocimiento* se encuentra en algunas aplicaciones de inteligencia artificial, pero su utilización es relativamente escasa. Casi nadie describe la sabiduría en el contexto de las aplicaciones del software para computadoras.

Los datos son información pura —una colección de hechos que es preciso procesar para que sean significativos—. La información se deriva asociando los hechos en el seno de un contexto dado. El conocimiento asocia la información obtenida en un contexto con otra información obtenida en un contexto distinto. Por Último, la sabiduría se produce cuando se derivan unos principios generalizados de conocimientos dispares. Estos

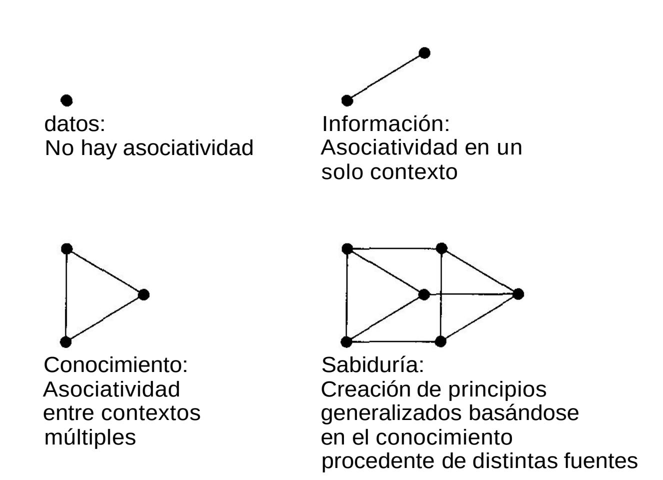


FIGURA 32.1. Un espectro de «información».

cuatro puntos de vista de la «información» se han representado esquemáticamente en la Figura 32.1.

Hasta la fecha, la gran mayoría del software que se ha construido ha tenido como misión procesar datos o información. Los ingenieros de software del siglo veintiuno seguirán estando preocupados por sistemas que procesen conocimiento[2]. El conocimiento es bidimensional. La información recogida acerca de una gama amplia de temas relacionados y no relacionados se agrupan para formar un cuerpo de hechos al que se denomina conocimiento. La clave es nuestra capacidad para asociar información de una gama de fuentes distintas que puedan no poseer conexiones evidentes entre sí y

[2] El crecimiento rápido de las tecnologías de minería de datos (*data mining*) y de almacenes de datos (*data warehousing*) reflejan este rápido crecimiento.

combinarlas de modo que nos proporcione algún beneficio definido.

Cita:
La sabiduría es el poder que nos capacita para utilizar el conocimiento en nuestro propio beneficio y en el de los demás.
Thomas J. Watson

Para ilustrar la progresión desde los datos al conocimiento, consideraremos los datos del censo que indican que el número de nacimientos en los Estados Unidos en **1996** fue de **4,9** millones. Este número representa un valor de datos. Al relacionar este dato con los nacimientos de los cuarenta años anteriores se puede extraer un elemento de información útil: aquellas personas que tuvieron muchos hijos en los años 50 y a principios de los **60,** están efectuando un último esfuerzo por tener niños antes de llegar al final de sus años fértiles. Esta información se puede conectar entonces con otros elementos de información aparentemente no relacionados, por ejemplo, el número actual de profesores de escuelas primarias que se retirarán durante la próxima década, el número de alumnos que cursarán estudios primarios y secundarios, o la presión habida sobre los políticos para reducir los impuestos y limitar por tanto los aumentos de sueldo para los profesores.

Todos estos elementos de información se pueden combinar para formular una representación del conocimiento -existirá una presión significativa sobre el sistema de educación de los Estados Unidos a principios del siglo veintiuno y esta presión proseguirá durante más de una década—. Mediante este conocimiento, puede aparecer la oportunidad de un negocio. Quizá existan oportunidades significativas para desarrollar nuevos modelos de aprendizaje que sean más efectivas y menos costosas que los enfoques actuales.

El futuro del software conduce a sistemas que procesan el conocimiento. Se ha estado procesando datos durante cincuenta años y extrayendo información durante casi tres décadas. Uno de los desafíos más significativos a los que se enfrenta la comunidad de la ingeniería del software consiste en construir sistemas que den el siguiente paso en el espectro: sistemas que extraigan el conocimiento de los datos y de la información para que sea práctica y beneficiosa.

32.6 LA TECNOLOGÍA COMO IMPULSOR

Las personas que construyen y utilizan software, el proceso de ingeniería del software que se aplica, y la información que se produce se ven afectados todos ellos por los avances en la tecnología del hardware y software. Históricamente, el hardware ha servido como impulsor de la tecnología de la computación. Una nueva tecnología de hardware proporciona un potencial. Entonces los constructores de software reaccionan frente a las solicitudes de los clientes en un intento de aprovechar ese potencial.

Las tendencias futuras de la tecnología del hardware tienen probabilidades de progresar en dos rutas paralelas. En una de las rutas, las tecnologías de hardware seguirán evolucionando rápidamente. Mediante la mayor capacidad que proporcionan las tecnologías de hardware tradicionales, las demandas efectuadas a los ingenieros de software seguirán creciendo.

Cita:
La nueva independencia electrónica recrea al mundo a imagen y semejanza del pueblo global.
Marshall McLuhan

Pero los cambios verdaderos de la tecnología de hardware se pueden producir en otra dirección. El desarrollo de arquitecturas de hardware no tradicionales (por ejemplo, máquinas masivamente paralelas, procesadores ópticos y máquinas de redes neuronales) pueden dar lugar a cambios radicales en la clase de software que se puede construir y también a cambios fundamentales en nuestro enfoque de la ingeniería del software. Dado que estos enfoques no tradicionales siguen estando en el primer segmento del ciclo de quince años, resulta difícil predecir la forma en que el mundo del software se modificará para adaptarse a ellas.

Las tendencias futuras de la ingeniería del software se verán impulsadas por las tecnologías de software. La reutilización y la ingeniería del software basada en componentes (tecnologías que todavía no están maduras) ofrecen la mejor oportunidad en cuanto a mejoras de gran magnitud en la calidad de los sistemas y se encuentran en el momento de comercializarse. De hecho, a medida que pasa el tiempo, el negocio del software puede empezar a tener un aspecto muy parecido al que tiene el negocio del hardware en la actualidad. Quizá existan fabricantes que construyan dispositivos diferentes (componentes de software reutilizables), otros fabricantes que construyan componentes de sistemas (por ejemplo, un conjunto de herramientas para la interacción hombre-máquina) e integradores de sistemas que proporcionen soluciones para el usuario final.

La ingeniería del software va a cambiar —de eso podemos estar seguros—. Pero independientemente de lo radicales que sean los cambios, podemos estar segu-

ros de que la calidad nunca perderá su importancia, y de que un análisis y diseño efectivos junto con una comprobación competente siempre tendrán su lugar en el desarrollo de sistemas basados en computadoras.

32.7 COMENTARIO FINAL

Ya han pasado 20 años desde que se escribió la primera edición de este libro. Y todavía me acuerdo de cuando era un joven profesor sentado en mi mesa y escribiendo (a mano) el manuscrito de un libro sobre un tema que no preocupaba a muchas personas y que aún menos entendían realmente. Recuerdo la cartas de los editores rechazando el libro y argumentando (de forma educada, pero contundente) que nunca habría mercado para un libro sobre «ingeniería del software». Afortunadamente, McGraw-Hill decidió intentarlo[3], y el resto ya es historia.

Durante los Últimos veinte años, este libro ha cambiado espectacularmente - e n ámbito, tamaño, estilo y contenido —. Al igual que la ingeniería del software misma, el libro también ha crecido, y por suerte también ha madurado durante los Últimos años.

Hoy en día un enfoque de ingeniería en el desarrollo del software de computadora es una sabiduría convencional. Aunque continúe el debate sobre el «paradigma adecuado», el grado de automatización y los métodos más efectivos, hoy en día los principios subyacentes de la ingeniería del software se aceptan a través de la industria. Entonces, ¿por qué solo hace muy poco tiempo que estamos siendo testigos de su gran adopción?

La respuesta, creo yo, radica en la dificultad de la transición de la tecnología y el cambio cultural que la acompaña. Aunque la mayoría de nosotros apreciamos la necesidad de una disciplina de ingeniería para el software, luchamos contra la inercia de la práctica anterior y nos enfrentamos con nuevos dominios de aplicaciones (y los que diseñan que son quienes trabajan en ellos) los cuales parecen estar preparados a repetir los errores del pasado.

Para facilitar la transición necesitamos muchas cosas —un proceso de software adaptable y sensible, métodos más eficaces, herramientas más potentes y una aceptación mejor de sus partidarios **y** soporte de los directores, y una dosis no pequeña de educación y «publicidad»—. La ingeniería del software no ha disfrutado de una gran publicidad, pero a medida que va pasando el tiempo el concepto se vende solo. De alguna manera, este libro es un «anuncio» para la tecnología.

Es posible no estar de acuerdo con todos los enfoques descritos en este libro. Algunas de las técnicas y opiniones son polémicas; otras deberán ajustarse para trabajar en diferentes entornos de desarrollo de software. Sin embargo, mi sincera esperanza es que *Ingeniería del Software: Un enfoque práctico* haya dibujado los problemas con los que nos enfrentamos; haya demostrado las ventajas de los conceptos de la ingeniería del software, y haya proporcionado un marco de trabajo de métodos y herramientas.

Como empieza un nuevo milenio, el software se ha convertido en el producto más importante de la industria y también más importante a nivel mundial. Su impacto e importancia han recorrido un largo camino. **Y,** sin embargo, una nueva generación de desarrolladores de software deberán encontrarse con muchos de los mismos retos con los que se enfrentaron generaciones anteriores. Esperemos que aquellas personas que se encuentren con el reto —ingenieros del software— tengan la sabiduría de desarrollar sistemas que mejoren la condición humana.

REFERENCIAS

[BOL91] Bollinger, T., y C. McGowen, «A Critical **Look** at Software Capability Evaluations,» *IEEE Software,* Julio de 1991, pp. 25-41.

[GIL96] Gilg, T., «What is level Six?», *IEEE Software*, Enero de 1996, pp. 97-98 y 103

[HOP90] Hopper, M.D., «Rattling SABRE, New Ways to Compete on Information», *Harvard Business Review,* Mayo-Junio de 1990.

[PAU93] Paulk, M. et al., *Capability Maturity Model for Software,* Software Engineering Institute, Camegie Mellon University, Pittsburgh, PA, 1993.

[3] Realmente los méritos son de Peter Freeman y Eric Munson, que fueron quienes convencieron a McGraw-Hill de que valdría la pena intentarlo.

PROBLEMAS Y PUNTOS A CONSIDERAR

32.1 Consiga una copia de las mejores revistas de negocios y de noticias de esta semana (por ejemplo: *Newsweek, Time, Business Week*). Haga una lista de todos los artículos o noticias que se puedan utilizar para ilustrar la importancia del software.

32.2 Uno de los dominios de aplicaciones de software más candentes son los sistemas y las aplicaciones basados en Web (Capítulo 29). Estudie la forma en que las personas, comunicación y proceso tienen que evolucionar para adoptar el desarrollo de WebApps de «próxima generación».

32.3 Escriba una breve descripción del entorno de desarrollo ideal de un ingeniero del software hacia el año 2010. Describa los elementos del entorno (hardware, software y tecnologías de comunicación) y su impacto en la calidad y el tiempo de comercialización.

32.4 Revise el estudio de los modelos de proceso evolutivo del Capítulo 2. Haga una investigación y recoja artículos recientes sobre este tema. Resuma las ventajas e inconvenientes de los paradigmas evolutivos basados en las experiencias indicadas en los artículos.

32.5 Intente desarrollar un ejemplo que comience con la recolección de datos puros y dé lugar a la adquisición de información, después del conocimiento, y finalmente de sabiduría.

32.6 Seleccione una tecnología de actualidad «candente» (no necesita ser una tecnología de software) que se esté estudiando en los medios populares y describa la forma en que el software posibilita su evolución e impacto.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Los libros que estudian las tendencias futuras del software y de la computación abarcan una amplia gama de temas técnicos científicos, económicos, políticos y sociales. Robertson (*The New Renaissance: Computers and The Next Level of Civilization*, Oxford University Press, 1998), argumenta que la revolución informática puede que sea el Único avance más significativo en la historia de la civilización. Dertrouzos y Gates (*What Will Be: How The New World of Information Will Change Our Lives*, Harperbusiness, 1998) proporcionan un profundo estudio de algunas de las direcciones que las tecnologías de la información pueden seguir en las primeras décadas de este siglo. Bamatt (*Valueware: Technology, Humanity and Organization*, Praeger Publishing, 1999) presenta un estudio intrigante de la «economía de las ideas» y de la forma en que se creará el valor económico a medida que evoluciona el ciber-negocio. El libro de Negroponte (*Being Digital*, Alfred A. Knopf, Inc., 1995) fue un best seller a mediados de los noventa y continúa proporcionando una visión interesante de la computación y de su impacto global.

Kroker y Kroker (*Digital Delirium*, New World Perspectives, 1997) han publicado una colección polémica de ensayos, poemas y humor en donde examinan el impacto de las tecnologías digitales en las personas y en la sociedad. Brin (*The Transparent Society: Will Technology Force us To Choose Between Privacy and Freedom?*, Perseus Books, 1999) repasa el debate continuo asociado a la pérdida inevitable de privacidad personal que acompaña al crecimiento de las tecnologías de la información. Shenk (*Data Smog: Surviving the Information Glut*, Harpercollins, 1998) estudia los problemas asociados con la «sociedad infectada de información» que se está produciendo del volumen de información que producen las tecnologías de información.

Miller, Michalski y Stevens (*21st Century Technologies: Promises and Perils of a Dynamic Future*, Brookings Institution Press, 1999) han publicado una colección de artículos y ensayos sobre el impacto de la tecnología en las estructuras sociales, de negocios y económicas. Para aquellos que estén interesados en estos temas técnicos, Luryi, Xu y Zaslavsky (*Future Trends in Microelectronics*, Wiley, 1999) han publicado una colección de artículos sobre las direcciones probables del hardware de computadora. Hayzelden y Bigham (*Software Agents for Future Communication Systems*, Springer Verlag, 1999) han editado una colección que estudia las tendencias del desarrollo de los agentes de software inteligentes.

Kurzweil (*The Age of Spiritual Machines, When Computers Exceed Human Intelligence*, Viking/Penguin Books, 1999) argumenta que dentro de 20 años la tecnología del hardware tendrá la capacidad de modelar completamente el cerebro humano. Borgman (*Holding on to Reality: The Nature of Information at the Turn of the Millenium*, University of Chicago Press, 1999) ha escrito una historia de información intrigante, haciendo un seguimiento de su papel en la transformación de la cultura. Devlin (*InfoSense: Turning Information into Knowledge*, W.H. Freeman & Co., 1999) trata de dar sentido al constante flujo de información que nos bombardea día a día. Gleik (*Faster: The Acceleration of Just About Everything*, Pantheon Books, 2000) estudia el ritmo cada vez más veloz del cambio tecnológico y de su impacto en todos los aspectos de la vida moderna. Jonscher (*The Evolution of Wired Life: From the Alphabet to the Soul-Catcher Chip-How Information Technologies Change Our World*, Wiley, 2000) argumenta que el pensamiento humano y la interacción trascienden la importancia de la tecnología.

Una variedad de fuentes de información sobre las tendencias futuras en la informática está disponible en Intemet. Una lista actualizada de referencias en la red se puede encontrar en **http://www.pressman5.com**

.

APÉNDICE

SIGLAS MÁS COMUNES EN INGENIERÍA DEL SOFTWARE

Siglas Español / Inglés

Término en Español	*Término equivalente en Inglés*
AAN (Análisis del área del negocio)	**BAA** (Business Area Analysis)
ACC (Autoridad de control de cambio)	**CCA** (Change Control Authority)
ACO (Acoplamiento entre clases de objetos)	**CBO** (Coupling Between Object Classes)
ACPs (Áreas clave de proceso)	**KPAs** (Key Process Areas)
ACV (Arquitectura del ciclo de vida)	**LCA** (Life Cycle Architecture)
ADOO (Análisis del dominio orientado a objetos)	**OODA** (Object-Oriented Domain Analysis)
AE (Análisis estructurado)	**SA** (Structured Analysis)
AG2D (Análisis geométrico de dos dimensiones)	**2DGA** (Two-dimensional Geometric Analysis)
AG3D (Análisis geométrico de tres dimensiones)	**3DGA** (Three-dimensional Geometric Analysis)
AOO (Análisis orientado a objetos)	**OOA** (Object-Oriented Analysis)
APD (Acceso público a datos miembros)	**PAD** (Public Access to Data members)
APH (Árbol de profundidad de herencia)	**DIT** (Depth of the Inheritance Tree)
API (Interfaz de programación de aplicaciones)	**API** (Application Programming Interface)
AROO (Análisis de los requisitos orientado a objetos)	**OORA** (Object-Oriented Requirements Analysis)
AVG (Análisis de valor ganado)	**EVA** (Earned Value Analysis)
AVL (Análisis de valores límites)	**BVA** (Boundary Value Analysis)
BDK (Kit de desarrollo Bean)	**BDK** (Bean Development Kit)
BRO (Operador relaciona1 y de ramificación)	**BRO** (Branch and Relational operator)
BS (brutos.sueldos)	**GW** (gross.wages)
C (Certificación)	**C** (Certification)
C (Consulta)	**Q** (Query)
C&I (construcción **e** integración)	**C&I** (Construction and Integration)
C/S (Cliente/servidor)	**C/S** (Client/server)
CAD (Diseño asistido **por** computadora)	**CAD** (Computer-Aided Design)
Cadena **DU** (cadena de definición-uso)	**DU** chain (Definition-Use chain)
CASE (Ingeniería del software asistida por computadora)	**CASE** (Computer-Aided Software Engineering)
CC (Centralizado controlado)	**CC** (Controlled Centralised)
CCM (Carencia de cohesión en los métodos)	**LCOM** (Lack **of** Cohesion in Methods)
CEU (Comprobación estadística de utilización)	**SUT** (Statistical Use Testing)
CFD (Cohesiones funcionales débiles)	**WFC** (Weak Function Cohesion)
CFF (Cohesiones funcionales fuertes)	**SFC** (Strong Functional Cohesion)
CGI (Interfaz común de pasarela)	**CGI** (Common Gateway Interface)
CGO (Capa de gestión de objetos)	**OML** (Object Management layer)
CK ***serie de métricas*** (Chidamber y Kemerer)	**CK** ***metrics suite*** (Chidamber and Kemerer metrics)
CN (Control numérico)	**NC** (Numerical **Control)**
CO (Complejidad de operación)	**OC** (Operation Complexity)
COCOMO (Modelo constructivo del coste)	**COCOMO** (Constructive Cost Model)
COI (Capacidad operativa inicial)	**IOC** (Initial Operational Capability)
COM (Modelo de objetos para componentes)	**COM** (Component Object Model)
CORBA (Arquitectura común de distribución de objetos)	**CORBA** (Common Object Request Broker Architecture)
CP (Control de periféricos)	**PC** (Peripheral Control)
CPTP (Coste de presupuestos del trabajo planificado)	**BCWS** (Budgeted Cost of Work Scheduled)
CPTR (Coste de presupuestos del trabajo realizado)	**BCWP** (Budgeted Cost of Work Performed)
CR (Conveniencia de la representación)	**LA** (Layout Appropriateness)
CRC (Clase-responsabilidad-colaboración)	**CRC** **(Class-Responsibility-Collaborator)**
CRTR (Coste real de trabajo realizado)	**ACWP** (Actual Cost of Work Performed)
CTA (Control de tráfico aéreo)	**ATC** (Air Traffic Control)
CYD (Comerciales ya desarrolladas)	**COTS** (Commercial Off-The-Shelf)
DBC (Desarrollo basado en componentes)	**CBD** (Component-Based Development)
DC (Descentralizado controlado)	**CD** (Controlled Decentralised)
DCBC (Descubrimiento de conocimiento en bases de datos)	**KDD** (Knowledge Discovery in Databases)
DCS (Diagrama de contexto del sistema)	**SCD** (System Context Diagram)
DD (Descentralizado democrático)	**DD** (Democratic Decentralized)
DDE (Desviación deliverada de la especificación)	**IDS** (Intentional Deviation from Specification)
DER (Diagrama de entidad-relación)	**ERD** (Entity Relationship Diagram)
DF (Diseño formal)	**FD** (Formal Design)

Término en Español	*Término equivalente en Inglés*
DFC (Despliegue de la función de calidad)	**QFD** (Quality Function Deployment)
DFC (Diagrama de flujo de control)	**CFD** (Control Flow Diagram)
DFD (Diagrama de flujo de datos)	**DFD** (Data Flow Diagram)
DFS (Diagrama de flujo del sistema)	**SFD** (System Flow Diagram)
DII (Documentación imprecisa *o* incompleta)	**IID** (Inaccurate or Incomplete Documentation)
DOO (Diseño orientado a objetos)	**OOD** (Object-Oriented Design)
DPR (Diseño para la reutilización)	**DFR** (Design for Reuse)
DRA (Desarrollo rápido de aplicaciones)	**RAD** (Rapid Application Development)
DSED (Desarrollo de sistemas estructurados de datos)	**DSSD** (Data Structured Systems Development)
DSJ (Desarrollo de sistemas Jackson)	**JSD** (Jackson System Development)
DSN (Diseño de sistema de negocio)	**BSD** (Business System Design)
DTE (Diagrama de transición de estados)	**STD** (State Transition Diagram)
EAIP (Entorno de apoyo integrado a proyectos)	**IPSE** (Integrated Project Support Environment)
EAT (Estructuras de análisis del trabajo)	**WBS** (**Work** Breakdown Structure)
EC (Especificación de control)	**CSPEC** (Control Specification)
ECS (Elementos de configuración del software)	**SCI** (Software Configuration Items)
ED (Elementos de datos)	**DE** (Data Elements)
EDC (Espacio de diseño cuantificado)	**QDS** (Quantity Design Space)
EDE (Elementos de datos de entrada)	**DEI** (Input Data Elements)
EDR (Elementos de datos retenidos)	**DER** (Retained Data Elements)
EDS (Elementos de datos de salida)	**DEO** (Output Input Data Elements)
EEC (Especificación de estructura de cajas)	**BSS** (Box Structure Specification)
EED (Eficacia de eliminación de defectos)	**DRE** (Defect Removal Efficiency)
EIE (Especificación incompleta **o** errónea)	**IES** (Incomplete or Erroneous Specification)
EIS (Entorno de ingeniería del software)	**SEE** (Software Engineering Environment)
ELD (Error en la lógica de diseño)	**EDL** (Error in Design Logic)
EP (Especificación de proceso)	**PSPEC** (Process Specification)
ERD (Error en representación de datos)	**EDR** (Error in Data Representation)
ES (Estados)	**ST** (States)
ETRP (Evaluación **y** técnica de revisión de programas)	**PERT** (Program Evaluation and Review Technique)
FA (Factor de acoplamiento)	**CF** (Coupling Factor)
FHA (Factor de herencia de atributo)	**AIF** (Attribute Inheritance Factor)
FHM (Factor de herencia de métodos)	**MIF** (Method Inheritance Factor)
FIR (FICA.impuesto.retenido)	**FTW** (FICA.Tax.Withheld)
FP (Factor de polimorfismo)	**PF** (Polymorphism Factor)
FPGC (Facilidades de presentación gráfica de computadora)	**CGDF** (Computer Graphics Display Facilities)
FURPS (Funcionalidad, facilidad de **uso,** fiabilidad, rendimiento y capacidad de soporte)	**FURPS** (Functionality, Usability, Reliability, Performance and Supportability)
FZ (Fijar zona)	**ZS** (Zone Set)
GBD (Gestión de base de datos)	**DBM** (Database Management)
GC (Generación de código)	**CG** (Code Generation)
GCS (Garantía de calidad del software)	**SQA** (Software Quality Assurance)
GCS (Gestión de la configuración del software)	**SCM** (Software Configuration Management)
GIP (Grupo independiente de prueba)	**ITG** (Independent Test Group)
OCM/OPM (Objetivo, cuestión (pregunta), métrica)	**GQM** (Goal, Question, Metric)
GTC (Gestión total de calidad)	**TQM** (Total Quality Management)
HD (Habilitar/deshabilitar)	**AD** (Arm-Disarm)*(p.* 731)
HIR (Hoja de información del riesgo)	**RIS** (Risk Information Sheet)
HTTP (Protocolo de transferencia de hipertexto)	**HTTP** (Hypertext Transfer Protocol)
IA (Inteligencia artificial)	**AI** (Artificial Intelligence)
IC (Inspección de código)	**CI** (Code Inspection)
1-CASE (CASE integrado)	**I-CASE** (Integrated CASE)
ICED (Índice de calidad de la estructura de diseño)	**DSQI** (Design Structure Quality Index)
ICMP (Protocolo de mensajes de control de Internet)	**ICMP** (Intemet Control Message Protocol)
IDC (Índice de desarrollo de coste)	**CPI** (Cost Performance Index)
IDL (Lenguaje de definición de interfaces)	**IDL** (Interface Definition Language)
IDP (Índice de desarrollo de planificación)	**SPI** (Schedule Performance Index)
IE (índice de error)	**EI** (Error Index)
IEC (Informes de estado de la configuración)	**CSR** (Configuration Status Reporting)
IEP (Incumplimiento de los estádandares de programación)	**VPS** (Violation of Programming Standards)
IES (índice de especialización)	**SI** (Specialisation Index)
IF (Índice de fase)	**PI** (Phase Index)
IGU (Interfaz gráfica de usuario)	**GUI** (Graphical User Interface)
IHM (Interfaz hombre-máquina)	**HCI** (Human-Computer Interface)
IMI (Interfaz de módulo inconsistente)	**ICI** (Inconsistent Component Interface)
IMS (índice de madurez del software)	**SMI** (Software Maturity Index)

Término en Español	*Término equivalente en Inglés*
IP (Protocolo de Intemet)	**IP** (Internet Protocol)
IPN (Ingeniería de proceso de negocio)	**BPE** (Business Process Engineenng)
IS (Ingeniería de sistemas)	**SE** (System Engineering)
ISBC (Ingeniería del software basada en componentes)	**CBSE** (Component-Based Software Engineering)
ISOO (Ingeniería del software orientada a objetos)	**OOSE** (Object-Oriented Software Engineering)
ISOOAC (Ingeniería del software orientada a objetos asistida por computadora)	**OOCASE** (Object-Oriented Computer Aided Software Engineering)
IU (Interfaz de usuario)	**UI** (User Interface)
IUSC (Interfaz de usuario y funciones de control)	**UICE** (User Interface and Control Facilities)
IWeb (Ingeniería Web)	**WebE** (Web-Engineering)
KLDC (mil líneas de código)	**KLOC** (thousands lines of code)
LDAs (Lenguajes de descripción arquitectónica)	**ADLs** (Architectural Description Languages)
LDC (Líneas de código)	**LOC** (Lines of Code)
LDP (Lenguaje de diseño de programas)	**PDL** (Program Design Language)
LIM (Lenguaje de interconexión de módulos)	**MIL** (Module Interconnection Language)
LPNI (Límite de proceso natural inferior)	**LPNL** (Lower Natural Process Limit)
LSPN (Límite superior del proceso natural)	**UNPL** (Upper Natural Process Limit)
MACA (Método de análisis de compromiso para la arquitectura)	**ATAM** (Architecture Trade-off Analysis Method)
MAD (Módulos de análisis de diseño)	**DAM** (Design Analysis Modules)
MCC (Mala interpretación de la comunicación del cliente)	**MCC** (Misinterpretation of Customer Communication)
MCC (Método del camino crítico)	**CPM** (Critical Path Method)
MCM (Modelo de capacidad de madurez)	**CMM** (Capability Maturity Model)
MCP (Marco común de proceso)	**CPF** (Common Process Framework)
MDOO (Métricas para el diseño orientado a objetos)	**MOOD** (Metrics for Object-Onented Design)
MEPS (Mejora estadística del proceso de software)	**SSPI** (Statistical Software Process Improvement)
MMCGP (Modelo de madurez de la capacidad de gestión de personal)	**PM-CMM** (People Management Capability Maturity Model)
MMGR (Mitigación, monitonzación y gestión del riesgo)	**RMMM** (Risk Mitigation, Monitoring and Management)
Modelo MOI (Motivación, organización, ideas o innovación)	**MOI Model** (Motivation, Organisation, Ideas or innovation)
MOM (Software intermedio orientado a mensajes)	**MOM** (Message Oriented Middleware)
MPC (Métodos ponderados por clase)	**WMC** (Weighted Methods per Class)
NCC (Número de clases clave)	**NKC** (Number of Key Classes)
NCR (Número de clases raíz)	**NOR** (Number of Root Classes)
NDD (Número de descendientes)	**NOC** (Number of Children)
NE (Número de escenarios)	**NSS** (Number of Scenario Scripts)
NN (Nodos de navegación)	**WoN** (Ways of Navigating)
NOA (Número de operaciones añadidas por una subclase)	**NOA** (Number of Operations Added by a subclass)
NOR (Número de operaciones redefinidas para una subclase)	**NOO** (Number **of** Operations Ovemdden by subclass)
NPD (Número de padres directos)	**FIN** (Fanin)
$\mathbf{NP}_{prom}$ (Número de parámetros por operación promedio)	$\mathbf{NP}_{avg}$ (Average Number of Parameters per operation)
NSUB (Número de subsistemas)	**NSUB** (Number of Subsystems)
OB (Objetos)	**OB** (Objects)
OCI (Orden de cambio de ingeniería)	**ECO** (Engineering Change Order)
OCV (Objetivos del ciclo de vida)	**LCO** (Life Cycle Objectives)
ODBC (Conectividad abierta de bases de datos)	**ODBC** (Open Database Connectivity)
OLCRS (Sistema interactivo para apuntarse a cursos)	**OLCRS** (On-Line Course Registration System)
OMG/CORBA (Grupo de gestión de objetos/Arquitectura común de distribución de objetos)	**OMG/CORBA** (Object Management Group/Common Object Request Broker Architecture)
OO (Orientado a objetos)	**OO** (Object-Oriented)
ORB (Agente de distribuición de objetos)	**ORB** (Object Request Broker)
P (Prueba)	**T** (test)
P&R (Pregunta y respuesta)	**Q&A** (Question and Answer)
PAT (Presupuesto a la terminación)	**BAC** (Budget At Completion)
PEI (Planificación de la estrategia de información)	**ISP** (Information Strategy Planning)
PEN (Proceso elemental de negocios)	**EBP** (Elementary Business Process)
PF (Puntos de función)	**FP** (Function Points) **y**
Pfu (Primitivas funcionales)	**FuP** (Functional Primitives)
PIE (Prueba incompleta o errónea)	**IET** (Incomplete or Erroneous Testing)
PMFu (Primitivas modificadas de función manual)	**FuPM** (Modified Manual Function Primitives)
POO (Programación orientada a objetos)	**OOP** (Object-Oriented Programming)
POP3 (Protocolo de oficina de correos versión **3)**	**POP3** (Post Office Protocol version **3)**
PP. (Planificación de prueba)	**TP** (Test Planning)
PPP (Porcentaje público y protegido)	**PAP** (Percent Public and Protected)
PRO/SIM (Construcción de prototipos y simulación)	**PRO/SIM** (PROtotyping and SIMulation)
PrOO (Pruebas orientadas a objetos)	**OOT** (Object-Oriented Testing)
PSI (Procesos secuenciales intercomunicados)	**CSP** (Communicating Sequential Processes)
RE (Relaciones)	**RE** (Relationships)

Término en Español	*Término equivalente en Inglés*
Re_i (Conexiones de relación)	**Re_i** (Relationship connections)
RFP	**RFP**
Rm (Rangos de movimiento)	**mR** (moving range)
RPC (Respuesta para una clase)	**RFC** (Response for a class)
RPN (Reingeniería de procesos de negocios)	**BPR** (Business Process Reengineering)
RR (Recolección de requisitos)	**RG** (Requirements Gathering)
RTF (Revisión técnica formal)	**FTR** (Formal Technical Review)
SBDOO (Sistemas de bases de datos orientadas a objetos)	**OODBS** (Object-Oriented Database System)
SCCT (Sistema de clasificación de cinta transportadora)	**CLSS** (Conveyor Line Sorting System)
SCE (Salidas/Control/entradas)	**OCI** (Output/Control/Input)
SDIU (Sistemas de desarrollo de la interfaz de usuario)	**UIDS** (User-Interface Development Systems)
SEI (Instituto de ingeniería de software)	**SEI** (Software Engineering Institute)
SGBDOO (Sistemas de gestión de bases de datos orientadas a objetos)	**OODBMS** (Object-Oriented Database Management System)
SGBDR (Sistema de gestión de bases de datos relacional)	**RDBMS** (Relational Database Management System)
SGH (Servicios de gestión de herramientas)	**TMS** (Tools management services)
SIG (Sistemas de información de gestión)	**MIS** (Management Information System)
SQA (Gestión de calidad de Software)	**SQA** (Software Quality Assurance
SQE (Ingeniería de Calidad del software)	**SQE** (Software Quality Engineenng)
SQL (Lenguaje de consultas estructurado)	**SQL** (Structure Query Language)
SSRB (Sistema de seguimiento y reparación de baches)	**PHTRS** (Pot Hole Tracking and Repair System)
T4G (Técnicas de cuarta generación)	**4GT** (Fourth Generation Techniques)
TADE (Técnicas de Análisis y diseño estructurado)	**SADT** (Structured Analysis and Design Technique)
TAP (Tabla de activación de procesos)	**PAT** (Process Activation Language)
TC (Tamaño de clase)	**CS** (Class Size)
TC_i (Muestras (tokens) de datos)	**TC,** (Data tokens)
TFEA (Técnicas para facilitar las especificaciones de la aplicación)	**FAST** (Facilitated Application Specification Techniques)
TI (Tecnologías de la información)	**IT** (Information technologies)
TLP (Error en la traducción del diseño al lenguaje de programación)	**PLT** (error in Programming Language Translation)
TMC (Tiempo medio de cambio)	**MTTC** (Mean-time-to-change)
TMEF (Tiempo medio entre fallos)	**MTTF** (Mean time to failure)
TMEF (Tiempo medio entre fallos)	**MTBF** (Mean time between failure)
TMO (Técnica de modelado de objetos)	**OMT** (Object Modelling Technique)
TMR (Tiempos medios de reparación)	**MTTR** (Mean time to repair)
TO_{prom} (Tamaño medio de operación)	**OS,,,** (Average Operation Size)
TR (Transiciones)	**TR** (Transitions)
UML (Lenguaje de modelado unificado)	**UML** (Unified Modelling Language)
USN (Unidad semántica de navegación)	**SNU** (Semantic Navigation Unit)
V&V (Verificación y validación)	**V&V** (Verification and Validation)
VC (Varianza del Coste)	**CV** (Cost Variance)
VC (Verificación de corrección)	**CV** (Correctness Verification)
VP (Varianza de planificación)	**SV** (Schedule Variance)
WebApps (Aplicaciones basadas en Web)	**WebApps** (Web-based Applications)
XML (Lenguaje de marcas extensible)	**XML** (Extensible Mark-up Language)

Siglas Inglés / Espanol

Término en Inglés	*Término equivalente en Español*
2DGA (Two-dimensional Geometnc Analysis)	**AG2D** (Análisis geométrico de dos dimensiones)
3DGA (Three-dimensional Geometric Analysis)	**AG3D** (Análisis geométrico de tres dimensiones)
4GT (Fourth Generation Techniques)	**T4G** (Técnicas de cuarta generación)
ACWP (Actual Cost of Work Performed)	**CRTR** (Coste real de trabajo realizado)
AD (Arm-Disarm)	**HD** (Habilitar/deshabilitar)
ADLs (Architectural Description Languages)	**LDAs** (Lenguajes de descripción arquitectónica)
Al (Artificial Intelligence)	**IA** (Inteligencia artificial)
AIF (Attribute Inbentance Factor)	**FHA** (Factor de herencia de atributo)
API (Application Programming Interface)	**API** (Interfaz de programación de aplicaciones)
ATAM (Architecture Trade-off Analysis Metbod)	**MACA** (Método de análisis de compromiso para la arquitectura)
ATC (Air Traffic Control)	**CTA** (Control de tráfico aéreo)
BAA (Business Area Analysis)	**AAN** (Análisis del área del negocio)
BAC (Budget At Completion)	**PAT** (Presupuesto a la terminación)
BCWP (Budgeted Cost of Work Performed)	**CPTD** (Coste de presupuestos del trabajo desarrollado)
BCWS (Budgeted Cost of Work Scheduled)	**CPTP** (Coste presupuestado del trabajo planificado)
BDK (Bean Development Kit)	**BDK** (Kit de desarrollo Bean)

Término en Inglés	*Término equivalente en Español*
BPE (Business Process Engineering)	**IPN** (Ingeniería de proceso de negocio)
BPR (Business Process Reengineering)	**RPN** (Reingeniería de procesos de negocios)
BRO (Branch and Relational operator)	**BRO** (Operador relacional y de ramificación)
BSD (Business System Design)	**DSN** (Diseño de sistema de negocio)
BSS (Box Structure Specification)	**EEC** (Especificación de estructura de cajas)
BVA (Boundary Value Analysis)	**AVL** (Análisis de valores límites)
C (Certification)	**C** (Certificación)
C&I (Construction and Integration)	**C&I** (construcción e integración)
C/S (Client/Server)	**C/S** (Cliente/servidor)
CAD (Computer-Aided Design)	**CAD** (Diseño asistido por computadora)
CASE (Computer-Aided Software Engineenng)	**CASE** (Ingeniería del software asistida por computadora)
CBD (Component-Based Development)	**DBC** (Desarrollo basado en componentes)
CBO (Coupling Between Object Classes)	**ACO** (Acoplamiento entre clases de objetos)
CBSE (Component-Based Software Engineering)	**ISBC** (Ingeniería del software basada en componentes)
CC (Controlled Centralised)	**CC** (Centralizado controlado)
CCA (Change Control Authority)	**ACC** (Autoridad de control de cambios)
CD (Controlled Decentralised)	**DC** (Descentralizado controlado)
CF (Coupling Factor)	**FA** (Factor de acoplamiento)
CFD (Control Fiow Diagram)	**DFC** (Diagrama de flujo de control)
CG (Code Generation)	**GC** (Generación de código)
CGDF (Computer Graphics Display Facilities)	**FPGC** (Facilidades de presentación gráfica de computadora)
CGI (Common Gateway Interface)	**CGI** (Interfaz común de pasarela)
CI (Code Inspection)	**IC** (Inspección de código)
CK metrics suite (Chidamber and Kemerer metrics)	**CK** serie de métricas (Chidamber y Kemerer)
CLSS (Conveyor Line Sorting System)	**SCCT** (Sistema de clasificación de cinta transportadora)
CMM (Capability Maturity Model)	**MCM** (Modelo de capacidad de madurez)
COCOMO (Constructive Cost Model)	**COCOMO** (Modelo constructivo del coste)
COM (Component Object Model)	**COM** (Modelo de objetos para componentes)
CORBA (Common Object Request Broker Architecture)	**CORBA** (Arquitectura común de distribución de objetos)
COTS (Commercial Off-The-Shelf)	**CYD** (Comerciales ya desarrolladas)
CPF (Common Process Framework)	**MCP** (Marco común de proceso)
CPI (Cost Performance Index)	**IDC** (Índice de desarrollo de coste)
CPM (Critica1 Path Method)	**CPM** (Método del camino crítico)
CRC (Class-Responsibility-Collaborator)	**CRC** (Clase-responsabilidad-colaboración)
CS (Class Size)	**TC** (Tamaño de clase)
CSP (Communicating Sequential Processes)	**PSI** (Procesos secuenciales intercomunicados)
CSPEC (Contrpl Specification)	**EC** (Especificación de control)
CSR (Configuration Status Reporting)	**IEC** (Informes de estado de la configuración)
CV (Correctness Verification)	**VC** (Verificación de corrección)
CV (Cost Variance)	**VC** (Varianza del Coste)
DAM (Design Analysis Modules)	**MAD** (Módulos de análisis de diseño)
DBM (Database Management)	**GBD** (Gestión de base de datos)
DD (Democratic Decentralized)	**DD** (Descentralizado democrático)
DE (Data Elements)	**ED** (Elementos de datos)
DEI (Input Data Elements)	**EDE** (Elementos de datos de entrada)
DEO (Output Data Elements)	**EDS** (Elementos de datos de salida)
DER (Retained Data Elements)	**EDR** (Elementos de datos retenidos)
DFD (Data Flow Diagram)	**DFD** (Diagrama de flujo de datos)
DFR (Design For Reuse)	**DPR** (Diseño para la reutilización)
DIT (Depth of the Inheritance Tree)	**APH** (Árbol de profundidad de herencia)
DRE (Defect Removal Efficiency)	**EED** (Eficacia de la eliminación de defectos)
DSQI (Design Structure Quality Index)	**ICED** (índice de calidad de la estructura de diseño)
DSSD (Data Structured Systems Development)	**DSED** (Desarrollo de sistemas estructurados de datos)
DU chain (Definition-Use chain)	Cadena **DU** (cadena de definición-uso)
EBP (Elernentary Business Process)	**PEN** (Proceso elemental de negocios)
ECO (Engineering Change Order)	**OCI** (Orden de cambio de ingeniería)
EDL (Error in Design Logic)	**ELD** (Error en la lógica de diseño)
EDR (Error in Data Representation)	**ERD** (Error en la representación de los datos)
EI (Error Index)	**IE** (Índice de error)
ERD (Entity Relationship Diagram)	**DER** (Diagrama de entidad-relación)
EVA (Earned Value Analysis)	**AVG** (Análisis de valor ganado)
FAST (Facilitated Application Specification Techniques)	**TFEA** (Técnicas para facilitar las especificaciones de la aplicación)
FD (Formal Design)	**DF** (Diseño formal)
FIN (Fanin)	**NPD** (Número de padres directos)
FP (Function Points)	**PF** (Puntos de función)
FTR (Formal Technical Review)	**RTF** (Revisión técnica formal)

Término en Inglés

FTW (FICA.tax.withheld)
FuP (Functional Primitives)
FuPM (Modified Manual Function Primitives)
FURPS (Functionality, Usability, Reliability, Performance and Supportability)
GQM (Goal, Question, Metric)
GUI (Graphical User Interface)
GW (gross.wages)
HCI (Human-Computer Interface)
HTTP (Hypertext Transfer Protocol)
1-CASE (Integrated CASE)
ICI (Inconsistent Component Interface)
ICMP (Intemet Control Message Protocol)
IDL (Interface Deñnition Language)
IDS (Intentional Deviation from Specification)
IES (Incomplete or Erroneous Specification)
IET (Incomplete or Erroneous Testing)
IID (Inaccurate or Incomplete Documentation)
IOC (Initial Operational Capability)
IP (Intemet Protocol)
IPSE (Integrated Project Support Environment)
ISP (Information Strategy Planning)
IT (Information Technologies)
ITG (Independent Test Group)
JSD (Jackson System Development)
KDD (Knowledge Discovery in Databases)
KLOC (thousands Lines of Code)
KPAs (Key Process Areas)
LA (Layout Appropriateness)
LCA (Life Cycle Architecture)
LCO (Life Cycle Objectives)
LCOM (Lack of Cohesion in Methods)
LOC (Lines of Code)
LPNL (Lower Natural Process Limit)
MCC (Misinterpretation of Customer Communication)
MIF (Method Inhentance Factor)
MIL (Module Interconnection Language)
MIS (Management Information System)
MIS (Miscellaneous)
MOI Model (Motivation, Organisation, Ideas or Innovation)
MOM (Message Oriented Middleware)
MOOD (Metrics for Object-Onented Design)
mR (moving Range)
MTBF (Mean Time Between Failure)
MTTC (Mean-Time-To-Change)
MTTF (Mean Time To Failure)
MTTR (Mean Time To Repair)
NC (Numerical Control)
NKC (Number Of key classes)
NOA (Number of Operations Added by a subclass)
NOC (Number of Children)
NOO (Number of Operations Overridden by subclass)
NOR (Number of Root classes)
NPavg (Average Number of Parameters **per** operation)
NSS (Number of Scenario Scripts)
NSUB (Number of Subsystems)
OB (Objects)
OC (Operation Complexity)
OCI (Output/Control/Input)
ODBC (Open Database Connectivity)
OLCRS (On-line Course Registration System)
OMG/CORBA (Object Management Group/Common Object Request Broker Architecture)
OML (Object Management Layer)
OMT (Object Modelling Technique)
OO (Object-Oriented)

Término equivalente en Español

FIR (FICA.impuesto.retenido)
Pfu (Primitivas funcionales)
PMFu (Primitivas modificadas de función manual)
FURPS (Funcionalidad, facilidad de uso, fiabilidad, rendimiento y capacidad de soporte)
GQM (Objetivo, cuestión, métrica)
IGU (Interfaz gráfica de usuario)
BS (brutos.sueldos)
IHM (Interfaz hombre-máquina)
HTTP (Protocolo de transferencia de hipertexto)
1-CASE (CASE integrado)
IMI (Interfaz de módulo inconsistente)
ICMP (Protocolo de mensajes de control de Internet)
IDL (Lenguaje de definición de interfaces)
DDE (Desviación deliverada de la especificación)
EIE (Especificación incompleta o errónea)
PIE (Prueba incompleta o errónea)
DII (Documentación imprecisa o incompleta)
COI (Capacidad operativa inicial)
IP (Protocolo de Intemet)
EAIP (Entorno de apoyo integrado a proyectos)
PEI (Planificación de la estrategia de información)
TI (Tecnologías de la información)
GIP (Grupo independiente de prueba)
DSJ (Desarrollo de sistemas Jackson)
DCBC (Descubrimiento de conocimiento en **bases** de datos)
KLDC (miles de líneas de código)
ACPs (Áreas clave de proceso)
CR (Conveniencia de la representación)
ACV (Arquitectura del ciclo de vida)
OCV (Objetivos del ciclo de vida)
CCM (Carencia de cohesión en los métodos)
LDC (Líneas de código)
LPNI (Límite de proceso natural inferior)
MCC (Mala interpretación de la comunicación del cliente)
FHM (Factor de herencia de métodos)
LIM (Lenguaje de interconexión de módulos)
SIG (Sistemas de información de gestión)
VAR (Varios)
Modelo MOI (Motivación, organización, ideas o innovación)
MOM (Software intermedio orientado a mensajes)
MDOO (Métricas para el diseño orientado a objetos)
Rm (Rangos de movimiento)
TMEF (Tiempo medio entre fallos)
TMC (Tiempo medio de cambio)
TMF (Tiempo medio de fallos)
TMR (Tiempos medios de reparación)
CN (Control numérico)
NCC (Número de clases clave)
NOA (Número de operaciones añadidas por una subclase)
NDD (Número de descendientes)
NOR (Número de operaciones redefinidas para una subclase)
NCR (Número de clases raíz)
NPprom (Número de parámetros por operación promedio)
NE (Número de escenarios)
NSUB (Número de susistemas)
OB (Objetos)
CO (Complejidad de operación)
SCE (Salidas/Dontrol/entradas)
ODBC (Conectividad abierta de bases de datos)
OLCRS (Sitema interactivo para apuntarse a cursos)
OMG/CORBA (Grupo de gestión de objetos/Arquitectura común de distribución de objetos)
CGO (Capa de gestión de objetos)
TMO (Técnica de modelado de objetos)
OO (Orientado a objetos)

Término en Inglés	Término equivalente en Español
OOA (Object-oriented Analysis)	**AOO** (Análisis orientado a objetos)
OOCASE (Object-Oriented Computer Aided Software Engineering)	**ISOOAC** (Ingeniería del software orientada a objetos asistida por computadora)
OOD (Object-Oriented Design)	**DOO** (Diseño orientado a objetos)
OODA (Object-Oriented Domain Analysis)	**ADOO** (Análisis del dominio orientado a objetos)
OODBMS (Object-Oriented Database Management System)	**SGBDOO** (Sistemas de gestión de bases de datos orientadas a objetos)
OODBS (Object-Oriented Database Systems)	**SBDOO** (Sistemas de bases de datos orientadas a objetos)
OOP (Object-Oriented Programming)	**POO** (Programación orientada a objetos)
OORA (Object-Oriented Requirements Analysis)	**AROO** (Análisis de los requisitos orientado a objetos)
OOSE (Object-Oriented Software Engineering)	**ISOO** (Ingeniería del software orientada a objetos)
OOT (Object-Oriented testing)	**PrOO** (Pruebas orientadas a objetos)
ORB (Object Request Broker)	**ORB** (Agente de distribuición de objetos)
OSavg (Average Operation Size)	**TOprom** (Tamaño medio de operación)
PAD (Public Access to Data members)	**APD** (Acceso público a datos miembros)
PAP (Percent Public and protected)	**PPP** (Porcentaje público y protegido)
PAT (Process Activation Language)	**TAP** (Tabla de activación de procesos)
PC (Peripheral Control)	**CP** (Control de periféricos)
PDL (Program Design Language)	**LDP** (Lenguaje de diseño de programas)
PERT (Program Evaluation and Review Technique)	**PERT** (Técnica de evaluación y revisión de programas)
PF (Polymorphism Factor)	**FP** (Factor de polimorfismo)
PHTRS (Pot Hole Tracking and Repair System)	**SSRB** (Sistema de seguimiento y reparación de baches)
PI (Phase Index)	**IF** (índice de fase)
PLT (error in Programming Language Translation)	**TLP** (Error en la traducción del diseño al lenguaje de programación)
PM-CMM (People Management Capability Maturity Model)	**MMCGP** (Modelo de madurez de la capacidad de gestión de personal)
POP3 (Post Office Protocol version 3)	**POP3** (Protocolo de oficina de correos versión)
PRO/SIM (PROtotyping and SIMulation)	**PRO/SIM** (Construcción de prototipos y simulación)
PSPEC (Process SPECification)	**EP** (Especificación de proceso)
Q (query)	**C** (Consulta)
Q&A (Question and Answer)	**P&R** (Pregunta y respuesta)
QDS (Quantity Design Space)	**EDC** (Espacio de diseño cuantificado)
QFD (Quality Function Deployment)	**DFC** (Despliegue de la función de calidad)
RAD (Rapid Application Development)	**DRA** (Desarrollo rápido de aplicaciones)
RDBMS (Relational Database Management System)	**SGBDR** (Sistema de gestión de bases de datos relacional)
RE (Relationships)	**RE** (Relaciones)
Rei (Relationship connections)	**Rei** (Conexiones de relación)
RFC (Response For a Class)	**RPC** (Respuesta para una clase)
RG (Requirements Gathering)	**RR** (Recolección de requisitos)
RIS (Risk Information Sheet)	**HIR** (Hoja de información de riesgos)
RMMM (Risk Mitigation, Monitoring and Management)	**RSGR** (Reducción, supervisión y gestión de riesgos)
SA (Structured Analysis)	**AE** (Análisis estructurado)
SADT (Structured Analysis and Design Technique)	**TADE** (Técnicas de Análisis y diseño estructurado)
SCD (System Context Diagram)	**DCS** (Diagrama de contexto del sistema)
SCI (Software Configuration Items)	**ECS** (Elementos de configuración del software)
SCM (Software Configuration Management)	**GCS** (Gestión de la configuración del Software)
SE (System Engineering)	**IS** (Ingeniería de sistemas)
SEE (Software Engineering Environment)	**EIS** (Entorno de ingeniería del software)
SEI (Software Engineering Institute)	**SEI** (Instituto de ingeniería de software)
SFC (Strong Functional Cohesion)	**CFF** (Cohesiones funcionales fuertes)
SFD (System Flow Diagram)	**DFS** (Diagrama de flujo del sistema)
SI (Specialisation Index)	**IES** (índice de especialización)
SMI (Software Maturity Index)	**IMS** (índice de madurez del software)
SNU (Semantic Navigation Unit)	**USN** (Unidad semántica de navegación)
SPI (Schedule Performance Index)	**IDP** (Indice de desarrollo de planificación)
SQA (Software Quality Assurance)	**GCS** (Garantía de calidad del software)
SQE (Software Quality Engineering)	**SQE** (Ingeniería de Calidad del software)
SQL (Structure Query Language)	**SQL** (Lenguaje de consultas estructurado)
SSPI (Statistical Software Process Improvement)	**MEPS** (Mejora estadística del proceso de software)
ST (States)	**ES** (Estados)
STD (State Transition Diagram)	**DTE** (Diagrama de transición de estados)
SUT (Statistical Use Testing)	**CEU** (Comprobación estadística de utilización)
SV (Schedule Variance)	**VP** (Varianza de planificación)
T (test)	**P** (Prueba)
TCi (Data tokens)	**TCi** (Muestras —tokens— de datos)
TMS (Tools Management Services)	**SGH** (Servicios de gestión de herramientas)
TP (Test Planning)	**PP** (Planificación de prueba)
TQM (Total Quality Management)	**GTC** (Gestión total de calidad)

Término en Inglés	*Término equivalente en Español*
TR (Transitions)	TR (Transiciones)
UI (User Interface)	IU (Interfaz de usuario)
UICE (User Interface and Control Facilities)	IUFC (Interfaz de usuario y facilidades de control)
UIDS (User-Interface Development Systems)	SDIU (Sistemas de desarrollo de la interfaz de usuario)
UML (Unified Modelling Language)	UML (Lenguaje de modelado unificado)
UNPL (Upper Natural Process Limit)	LPNS (Límite de proceso natural superior)
V&V (Verification and validation)	V&V (Verificación y validación)
VPS (Violation of Programming Standards)	IEP (Incumplimiento de las estádandares de programación)
WBS (Work Breakdown Structure)	EAT (Estructuras de análisis del trabajo)
WebApps (Web-based Applications)	WebApps (Aplicaciones basadas en Web)
WebE (Web-Engineenng)	IWeb (Ingeniería Web)
WFC (Weak Function Cohesion)	CFD (Cohesiones funcionales débiles)
WMC (Weighted methods per class)	MPC (Métodos ponderados por clase)
WoN (Ways of navigating)	NN (Nodos de navegación)
XML (Extensible Mark-up Language)	XML (Lenguaje de marcas extensible)
ZS (Zone Set)	FZ (Fijar zona)

ÍNDICE

Ingeniería del Software

Un enfoque práctico

Quinta edición

Roger S. PRESSMAN

¿Está estudiando Ingeniería del Software? ¿Necesita un libro de texto?

Ingeniería del Software. **Un enfoque práctico** es el libro ideal para apoyar sus estudios dada la experiencia con la que cuenta. En esta quinta edición se ha realizado una revisión completa con objeto de reflejar las prácticas más actuales de la ingeniería del software. Se ha incluido material sobre comercio electrónico, Java y UML, y materias tales como formulación, análisis y pruebas de aplicaciones Web en un capítulo nuevo sobre ingeniería Web.

Con la adaptación especial realizada por Darrel Ince, este es el libro ideal para estudiantes de ingeniería del software y de electrónica. También resultará un libro atrayente para profesionales de industria que buscan tener una buena guía dentro de la ingeniería del software.

Características clave:
- Últimas versiones de Java y UML.
- Tratamiento más amplio de sistemas distribuidos incluyendo seguridad y comercio electrónico.
- Examen del entorno Web y sus implicaciones en la ingeniería del software.
- Nuevos estudios de casos y ejemplos para demostrar la puesta en práctica de esta teoría.

Visite la nueva página Web de Pressman **http://www.pressman5.com** para obtener más información acerca de:
- Guías de estudio.
- Estudios de casos.
- FAQ (preguntas más frecuentes).
- Preguntas de opción múltiple.
- Referencias a otras informaciones.

__Roger S.Pressman__ es una autoridad de reconocido prestigio internacional en el desarrollo de mejoras dentro del proceso del software y en tecnologías de ingeniería del software. Actualmente, es el presidente de Pressman & Associates, Inc., una consultoría especializada en prácticas de ingeniería del software.

__Darrel Ince__ es profesor de Informática en la Open University y autor de una amplia bibliografía.



http://www.mcgraw-hill.es

McGraw-Hill Interamerican
de España, S. A. U.
A Subsidiary of The McGraw-Hill Compan

ISBN: 84-481-3